THE UNIVERSITY *of* EDINBURGH

ALGORITHMS, ABSTRACTION AND IMPLEMENTATION:

A Massively Multilevel Theory of

Strong Equivalence of Complex Systems

Carol Lynn Foster

Ph.D.

University of Edinburgh

1990

For my parents

ACKNOWLEDGEMENTS

One way to look at this work is as the completion and formalisation of my view of levels and algorithms, a view that grew out of a computer science education followed by years of working with a variety of complex computer systems. My pragmatic and imprecise model was revised and clarified by the philosophical and psychological perspective of the course, discussions and readings at the Centre for Cognitive Science at the University of Edinburgh. I was able to see how the developing theory fit back into computer science while lecturing at Massey University in New Zealand as I completed the thesis.

I am especially grateful to my supervisors, Keith Stenning and Jon Oberlander, who provided just the right balance of restraint and freedom to go my own way, often across great distances. (Hurrah for email!) Their encouragement and comments were invaluable. Thanks are also in order to others who took the trouble to read and critique one or more draft chapters. In particular, Dean Allemang, Peter Dayan, John Hudson, John Podd, Barbara Scholz, Randall Stark and Kim Sterelny raised thought-provoking questions. (Of course, they incur no responsibility for the final product.) In addition, colleagues, friends and relatives too numerous to mention have helped in ways too numerous to mention, from wide-ranging discussions to cooking meals to posting books around the world.

This research was supported for three years by an Edinburgh University Studentship and an Overseas Research Scheme (U.K.) award. It also drew on the resources of the Centre for Cognitive Science at the University of Edinburgh, the Department of Computer Science at Massey University (N.Z.), and the ESRC funded Human Communication Research Centre (HCRC).

ABSTRACT

This thesis puts forward a formal theory of levels and algorithms to provide a foundation for those terms as they are used in much of cognitive science and computer science. Abstraction with respect to concreteness is distinguished from abstraction with respect to detail, resulting in three levels of concreteness and a large number of algorithmic levels, which are levels of detail and the primary focus of the theory.

An algorithm or ideal machine is a set of sequences of states defining a particular level of detail. Rather than one fundamental ideal machine to describe the behaviour of a complex system, there are many possible ideal machines, extending Turing's approach to reflect the multiplicity of system descriptions required to express more than weak input-output equivalence of systems. Cognitive science is concerned with stronger equivalence; e.g., do two models go through the same states at some level of description? The state-based definition of algorithms serves as a basis for such strong equivalence and facilitates formal renditions of abstraction and implementation as relations between algorithms. It is possible to prove within the new framework whether or not one given algorithm is a valid implementation of another, or whether two unequal algorithms have a common abstraction, for example. Some implications of the theory are discussed, notably a characterisation of connectionist versus classical models.

TABLE OF CONTENTS

CHAPTER I
INTRODUCTION AND STARTING POINT

What is an algorithm? When are two algorithms the same? This thesis began as an attempt to characterise and contrast connectionist and classical algorithms and became an investigation into the more fundamental notion of 'algorithm' itself.

There are easy answers for the above questions from theoretical computer science and logic, though even these definitions are surprisingly diverse and informal. The most precise account in those contexts describes an algorithm as a recipe, or a finite list of instructions (sometimes constrained to halt after a finite number of steps) written in a formal language such as LISP or FORTRAN. Two such recipes or programs are weakly equivalent if they produce the same output from the same input. This weak input-output (or extensional) equivalence stands in contrast to the strong equivalence of the title of this thesis.

For cognitive scientists, cognitive psychologists and even for many computer scientists, however, this cannot be the whole story. We speak of connectionist algorithms for pattern recognition, for example, although there may not be any obvious list of instructions in such algorithms. Students of programming are often asked to implement a particular algorithm (such as 'binary search' or 'heap sort') in a language of their choice. This makes no sense unless there is some idea of 'algorithm' that is independent of any particular language. Weak input-output equivalence is clearly not enough to classify algorithms if different kinds of searches (finding one item among many) and sorts (ordering of input items) are to be distinguished. In fact, it has been argued that psychology is primarily concerned with algorithms,

characterised as 'how' a system does what it does rather than just 'what' the system does (see Chapter III). Clarifying the 'how' in terms of the states through which a system passes as input is transformed to output forms the basis of the theory of strong equivalence.

The main contribution of this thesis is a formal definition for the cognitive scientists' and practical computer scientists' sense of algorithm independent of its implementation or realisation in a particular language, architecture or hardware. This definition facilitates accounts of strong equivalence and multiple instantiability, and is at the same time related to logical recipes in an interesting way. An algorithm is taken to be a set of sequences of states; this set also defines an ideal machine and a level of description.

Rather than one fundamental ideal machine (such as the Turing machine that underwrites theoretical computer science), there are many possible ideal machines at different levels of detail, providing the flexibility to accommodate the many possible levels of description of complex systems. Cognitive science requires more than equivalence of input-output profiles of systems or models, more even than equivalence of time and space requirements on some canonical machine. It needs a means of expressing equivalence in this stronger sense: do the systems or models go through the same states at any level of description? The state-based definition of algorithms that is the heart of this thesis captures that strong equivalence: two algorithms are the same if they contain the same sequences of states; two sequences are the same if they contain the same states in the same order. The new definition of algorithms does not stand alone but is part of a larger formal framework that includes definitions of implementation and abstraction as relations between states, sequences and algorithms.

Taken together these provide a well-defined theory of levels of description of complex systems. This makes it possible, for example, to prove whether or not one algorithm may be viewed as an implementation of another algorithm or whether two algorithms that are unequal at a given level of detail can be shown to have a common abstraction at a coarser level of detail.

This chapter, 'Introduction and Starting Point', sets the stage in terms of assumptions and approach. Then the main thesis is gradually developed and its implications considered in chapters which themselves contain significant related contributions.

In Chapter II, 'Levels', the received version of levels of description based on Marr (1982) is reviewed along with recent variations. This critical examination culminates in a more rigorous theory of levels, a necessary ingredient for a multilevel framework for algorithms. In the new definition of levels, abstraction and implementation in terms of concreteness are distinguished from abstraction and implementation in terms of detail, leaving three levels of concreteness. These are 'the mathematical level' best characterised by lack of space or time measures, 'the physical level' characterised by absolute space and time and, between these two, 'the algorithmic level' characterised by relative space and time measures. In addition to these levels of concreteness, there is a very large number of levels of detail, all considered to be part of the algorithmic level of concreteness. An algorithmic level description can be thought of as a description of a physical system that may give more than just a mathematical input-output relationship; in general, it says something about the states through which the system passes or could pass.

It is this latter sort of level that is the primary focus of the thesis. In Chapter III, 'Algorithms', existing definitions of algorithms are surveyed, along with the usual means of comparing them. This is followed by the introduction of the new definition of algorithms in terms of states and sequences. As a first cut, abstraction and implementation are defined for algorithms as procedural abstraction operations, such as 'selection of sequences' or 'change of labels'. Throughout Chapter II exclusive-or is used as a simple example to illustrate the issues; this practice is continued in Chapter III which concludes with a detailed and novel comparison of five different exclusive-or algorithms (connectionist and otherwise) within the new framework.

Chapter IV extends this practical approach to 'Quicksort: A More Detailed Example'. Besides giving a feel for the scope and utility of the new framework, an unusual analysis of Quicksort results. At some levels of detail not ordinarily considered, this algorithm that was originally developed to be quick on sequential von Neumann machines looks surprisingly suggestive of a realisation describable as parallel. If the essence of Quicksort lies in a common abstraction of algorithms implied by textbook procedures, then that essence is likely to be obscured rather than clarified by those textbook procedures.

Chapter V, 'Algorithms Formally Revisited', returns to give a more complete and rigorous rendition of the definitions of algorithms, abstraction and implementation. This formalisation is modelled closely on theoretical computer science, particularly the formulation of Davis and Weyuker (1983) who give the most precise definition of algorithm in the survey of Chapter III. It is possible to derive theorems from the definitions, and some simple theorems are proved. The

relationship between algorithms and Turing computability is considered, and a proof of a (very) restricted recasting of Church's Thesis is within reach, given that algorithms have been defined more generally than for particular languages; such a proof is outlined. In addition, it is shown that the rather unconstrained approach to multilevel algorithms is not so liberal as to be meaningless: proofs are given to show that not every algorithm can be seen as an implementation of every other algorithm and not every sequence can be seen as an implementation of every other sequence, even if they both have the same number of states.

The notions of levels and algorithms are so pervasive and fundamental to cognitive science and computer science that the potential implications of the first chapters are widespread. Chapter VI, 'Some Implications', includes a sampling from a cognitive science perspective. Any theory about complex systems that can be put into formal algorithmic terms, such as those defined in Chapter V, is likely to be seen in sharper (or at least edifyingly different) focus. Programme and process explanations (Jackson and Pettit, 1988) and representational redescription (Clark and Karmiloff-Smith, forthcoming) come under the spotlight, for example.

Also of note in the varied terrain of Chapter VI is a return to the original motivation behind this research: what is the relationship between connectionist and classical algorithms? The main distinctions that show up in the new framework are (1) a greater degree of indeterminacy and (2) a greater degree of distributed change, i.e., number of values changing in an individual state change, in connectionist systems as opposed to classical ones -- at their usual levels of description. In addition, connectionist systems tend to be viewed as having more (and more detailed) levels of theoretical

interest. However, it is the similarity rather than the difference that emerges most strongly by locating algorithms in this formalism. Any system at all can be seen as a sequence of states with different explicit representations and different degrees of nondeterminism and distributed change, depending on the level of detail of its algorithmic description. This thesis can be seen as the long road to grounding the apparently familiar terms of such a characterisation.

Personal Perspective

I came to cognitive science as a computer scientist who knew something about linguistics and who wanted to know more about psychology. I did not expect to encounter such familiar constructs as I did in flow-charted theories, e.g. Wason and Johnson-Laird's (1972) analysis of the four-card problem, and database-like lexical access methods. While it was reassuring to be on familiar ground, something seemed to be missing, something akin to the now standard arguments against classical computational theories of psychology (see, for example, 'The Appeal of PDP' in Rumelhart and McClelland et. al., 1986). In particular, the grain of beliefs and goals in terms of things like hamburgers and restaurants (Schank and Abelson, 1977) seemed too coarse. Dennett (1978, p. 46) makes this point:

> Representations apparently play roles at many different levels in the operation of the brain. I have already mentioned the possibility of codes representing information about the tension of eye muscles and so forth, and these representations do not fall into the class of our beliefs,

and (on the same page)

> The conclusion is that writing -- for instance, brain writing -- is a dependent form of information storage. The brain must store at least some of its information in a manner not computable by a brain-writing model...

Ignoring such low level views of representations is just the problem with conclusions such as Fodor's (Fodor, 1975; Fodor et. al., 1980) that we are born with language-of-thought words for, say, compact discs or video cassette recorders. In a way this is ridiculous, yet in a way it is not; we may well have from birth the potential, perhaps implicit, building blocks of such concepts. Surely computational psychology is not limited to trafficking only in high-level concepts common in current natural language. In fact, the better-understood complex systems of computer science seem to provide just the right domain in which to investigate and clarify many-levelled and implicit representation. As Fodor and Pylyshyn (1988) are at pains to point out, these systems are also parallel at some levels of description. Still, we speak of explicit data structures such as stacks and lists.

The resurgence of connectionism provided timely and vivid illustrations of systems in which high level concepts depend on lower level ones in complicated (not straightforwardly hierarchical) ways. The attempt to disentangle what is different and appealing about connectionism led inevitably to consideration of levels and algorithms. These were two areas about which I had to be clear before I could progress. The development of a precise and cohesive theory of levels and algorithms became the major project and subject of this thesis.

Assumptions and Approach

Questions about the relationships between levels and algorithms, hardware and software, implicit and explicit representations or programmed and emergent behaviour can be tricky even when posed with respect to the simplest program or system imaginable. For that reason I concentrate on exclusive-or; most of the main points of the thesis can be made using this easy example. The questions are indeed still difficult and, I claim, the answers are interesting and general. Consequently, my approach has been to use simple and uncluttered computational examples as far as possible to develop a formal theory of algorithms, ignoring as many other interacting and thorny philosophical problems as I can. In Chapter II there is some attempt to consider theories of levels that are based on definitions of semantics and reality, but such distinctions are minimised in the final theory.

It may therefore be easier to say what this thesis is not about. It does not say anything about 'building a thinker' or 'instantiation' of mental states, to borrow the terminology of Clark (1989). He contrasts (p. 177)

> the project of psychological explanation (laying out the computational causes of intelligent behavior) and that of instantiation (making a machine that actually has thoughts).

While a causal story can be told in terms of the new theory, it is best seen as a framework for describing a complex system's behaviour at varying levels of detail. What counts as a description as opposed to an explanation can depend on the context and the question at hand. For example, consider the context of a calculator that adds two integers between 0 and 20, always coming up with the

right answer except for the case where '2' and '2' are entered. Then its response is '5'. At a more detailed level of description, one that includes interim states of the calculator and a closer look at its internal mechanisms, the system might be described as performing a table lookup. Assuming that this level of description includes the central table, it might be seen to contain the entry '5' for the indices '2' and '2', providing an explanation for the behaviour described above.

Given a second system that also adds up two integers between 0 and 20, except that it too responds with '5' when given '2' and '2', I would say that the two systems under these descriptions are weakly (or input-output, or extensionally) equivalent, or predictively equivalent. That is, given the input-output behaviour of one system, the input-out behaviour of the other can be predicted. The two systems may or may not be strongly equivalent, or go through the same interim states, at a more detailed level of description. If, at some particular level, they are strongly equivalent in this sense, then one can be said to simulate the other (at that level of detail). Again, note that the assertion that one system simulates another, at no matter how detailed a level, makes no claim that the simulation (or model) instantiates the states of the simulated. In talking about weak versus strong equivalence, I will specify the level of detail of the description as far as possible. Once again the need for a precise meaning of such a level reveals itself.

I am aiming for a theory of levels and algorithms that does not depend on principles of semantics or intentionality or broad content. This is partly a cautious move, given the state of the art; these issues are among the most difficult in cognitive science. It also coincides with the goal of simplicity. In any case it is interesting to see how far we can get without them.

The same holds for causality. Of course causality cannot be omitted from all explanation, but distinguishing (potentially very detailed) descriptions of a process from causal claims isolates the descriptive issues still further. Yet another major area about which I shall have little to say is development, in both the sense of individual learning and that of evolutionary development of a species. Here again there are interesting and important issues for describing and explaining complex systems, but they too often distract from a clear view of levels of description of a system at a particular stage in its development.

I am assuming materialism throughout though I shall not argue for it. As Dennett asserts (1978, p. 83),

> If Church's Thesis is correct, then the constraints of mechanism are no more severe than the constraint against begging the question in psychology, for any psychology that stipulated atomic tasks that were 'too difficult' to fall under Church's Thesis would be a theory with undischarged homunculi.

If we do not abstract away from the finiteness of physical systems, then accepting something like Church's Thesis becomes even less problematic (see Chapter V).

In keeping with materialism and simulation, and avoiding semantics and causality, I use 'computation' and 'representation' in a very general sense (at least as general as P.S. Churchland, 1986, and Clark, 1989, and in contrast to Putnam, 1983a, and Pylyshyn 1984, e.g.), unrestricted by von Neumann architecture or a need to match intentional terminology. Algorithms turn out to be a generalisation of 'computations', as the term occurs in theoretical computer science (see Chapter III for full details), to sequences of states at many possible levels

of detail. Representations are labelled values in those states, at whatever level of description. States are taken to be primary, with processes only implicit. While there is some discussion of this in the text, it is best viewed as an assumption.

CHAPTER II
LEVELS

MARR'S LEVELS

Perhaps the most frequently cited terminology for levels of explanation is that of David Marr (Marr, 1977; Marr, 1982; Marr and Poggio, 1977). As a starting point therefore I will first present a somewhat simplified version of Marr's levels. This version will then be related to a rather different interpretation, following the original sources more closely. (Note that 'algorithm' and 'implementation' will be used in their usual vague senses until the new definitions are introduced.)

An Example

To illustrate the concepts and definitions of levels, consider the following example, which will be used throughout this chapter and the next. It is meant to be a simple example of a complex system. Imagine a standard computer, say a VAX/750, that has been programmed in PASCAL to calculate the function exclusive-or. This function is just the mapping that takes the ordered pair (0,0) to 0,(0,1) to 1,(1,0) to 1 and (1,1) to 0, or it could be described in words as '"A exclusive-or B" is true just in case either "A" or "B" is true, but not both.' This function is often abbreviated by 'xor', which I will use interchangeably with 'exclusive-or'. Also imagine that the PASCAL program has been compiled, assembled and executed (that is, it has been translated into the lowest level of machine language necessary for it to be run, and then run), so that the VAX appears to the user at the terminal as an exclusive-or calculator. If a '1' and then a '0' are entered, a '1' comes back. If a '0' and a '0' are entered, a '0' appears on the screen, and so forth.

A Simplified View

I am tempted to call this section 'The Standard View', because what I will be describing is what I understood to be Marr's levels from discussions and reading in cognitive science in general, without trying to sort out what Marr actually had in mind. This appears to be a common misconception. See, for example, the view of Broadbent (and to a lesser extent, McClelland and Rumelhart) in 'Other Views of Levels' later in this chapter. Peacocke (1986) comments in a footnote (pp. 102) that 'There is an element of construction in saying that, as far as the function goes, it is only the function in extension which is relevant to Marr's level 1' and Clark (1990, p. 284) also notes the discrepancy between Chomsky's competence theory, on which Marr's level 1 or computational level is based, and the 'official dogma'. I have included the simplified version here because I still think it is clear and relatively uncontroversial. In fact, the top level will be preserved intact in the final theory of this thesis.

Marr suggested three levels of explanation for complex systems. The top, or most abstract, level is called the computational level. In our example, this is the exclusive-or function as a mathematical object. At this level, we can prove things about the function itself, such as A xor B = (A or B) and (not(A and B)), independently of how (or even if) that function may be implemented in some physical device using some algorithm.

The middle level is the algorithmic level. At this level, there can be an explanation of how the complex system produces the appropriate output from the given

input. The PASCAL program mentioned in the example embodies such an algorithm, as in the following:

```
GET X;
GET Y;
IF X = Y THEN
     Z = 0;
OTHERWISE
     Z = 1;
OUTPUT(Z);
END;
```

(This is not PASCAL, but a semiformal pseudocode.) Of course, this example does not give the only possible algorithm, which is why the computational level may be (some would say should be) considered separately. An alternative algorithm might be this one:

```
GET Y;
GET X;
OUTPUT((X OR Y) AND (NOT(X AND Y))).
```

At the bottom, most concrete, level is the implementation level. This is the level of explanation in terms of the physical device or hardware in which an algorithm is implemented. An explanation of the VAX hardware and how it functions to produce the appropriate behaviour for exclusive-or would be an explanation at this level. As was the case in going from the computational to the algorithmic level, the move from the algorithmic to the implementation level involves a narrowing down to one of many alternatives. Our PASCAL program could run on a VAX or an IBM personal computer. As Searle points out (1980), any function that is Turing-computable can be implemented 'using a roll of toilet paper and a pile of small stones' (p. 301 in Mind Design, referring to Weizenbaum, 1976).

Marr's View

What Marr actually says is rather different, particularly with respect to the computational level. On the other hand, the implementational level, or 'hardware implementation level' as he calls it (Marr 1982, p. 25), is much the same as in the simplified view. This is just the level of explanation of the physical device in which an algorithm is implemented, be it person or computer. As far as it goes -- this level is not given much attention by Marr -- it is straightforward (but I will have more to say about this later). One point that he emphasises about the relationship between this level and the algorithmic level is that the former places constraints on the latter, restricting the representations and operations that can be used at the algorithmic level.

The 'representation and algorithm level' (Marr's name for the algorithmic level) can be seen as being squeezed between the constraints of the hardware from below and the constraints of the computation to be performed from above. Once again, the simplified view gives a good idea of what Marr means by this level, but there are some issues which are stressed in Marr's rendition but are not usually mentioned by others. These issues are centred around the nature and primacy of the representation. The representation is itself a system, much like a grammar: it is symbolic and compositional. 'A representation,' as Marr defines it (Marr 1982, p. 20), 'is a formal system for making explicit certain entities or types of information, together with a specification of how the system does this.' He gives as examples numeral systems, such as binary and Roman numerals, pointing out that particular representations aid or impede various operations, such as determining whether or not a given

number is a power of two. The distinction between process (algorithm) and representation is crucial, and the choice of representation comes first and restricts the choice of process. Of course, there may still be many possible suitable algorithms, given a particular representation.

While a description at the representation and algorithm level is meant to answer the question 'how?', the computational level description is intended to answer the questions 'what?' and 'why?'. Unfortunately, a precise definition of Marr's all-important computational level and even his more general term 'information processing task' are left frustratingly vague. One of the most unambiguous things he says about the computational level is that it is the same level as Chomsky's competence theory for transformational grammar (Marr, 1977, p. 38), referring to Chomsky's Aspects of the Theory of Syntax.

Chomsky (1965, e.g.) distinguishes competence theories from performance theories in linguistics, and he considers this to generalise to 'empirical investigation of other complex phenomena' (p. 4). A fundamental feature of a competence theory is its idealisation (p. 3):

> Linguistic theory is concerned primarily with an ideal speaker-listener, in a completely homogeneous speech-community, who knows its language perfectly and is unaffected by such grammatically irrelevant conditions as memory limitations, distractions, shifts of attention and interest, and errors (random or characteristic) in applying his knowledge of the language in actual performance.

It is clear that this is different from the top level as described above in 'The Simplified View'.

If we take the competence theory as a description of an individual system (but not necessarily of the purpose for which it was intended), the following observations will be important in the formulation of a more rigorous definition of levels. A competence theory of an ideal speaker will be very likely to have a different input-output profile to a normal human speaker. In the same way, a competence theory of an ideal exclusive-or calculator might have a different extension to an exclusive-or calculator with bugs while under development. At the same time, both the competence theories and the mathematical extensions can be implemented in an arbitrary member of possible algorithms or hardware devices. In any case, the controversy and confusion surrounding Chomsky's competence-performance distinction argues against its inclusion in a clear-cut levels framework.

Let us see if Marr makes the distinction any clearer. In his own description of the computational level in Vision, he states that the

> important features are (1) that it contains separate arguments about what is computed and why and (2) that the resulting operation is defined uniquely by the constraints it has to satisfy (p. 23).

His example of a cash register's computational theory has the general theory of addition as the 'what' and the relevance of this theory to the cash register as 'why'. For example,

> If you buy nothing, it should cost you nothing; and buying nothing and something should cost the same as buying just the something. (The rules for zero.) (p. 22),

and so on for commutativity, associativity and inverses. These rules also serve to illustrate the constraints of the second feature, given in (2) above.

I think that it is clear that in the computational level, Marr, like Chomsky, intends more than the purely mathematical view of an abstract function. This is best illustrated by his assertion that the computational problem of chess is more than 'take the opponent's king' (Marr, 1977, p. 38). Yet there remains an element of abstraction and idealism in this level: it is independent of any particular representation, as addition is independent of the numeral system. Neither is the computational level description an existence proof; evidence that some effective procedure, any effective procedure, exists by which the problem under consideration can be solved is not enough. In a footnote on the same page, Marr states

> One computational theory that in principle can solve chess is exhaustive search. The real interest lies however in formulating the pieces of computation that we apply to the game. One presumably wants a computational theory that has a rather general application, together with a demonstration that it happens to be applicable to some class of games of chess, and evidence that we play games in this class.

This is Marr's most explicit textual evidence that the computational level is really a coarse algorithmic level, or maybe a theory at level 1.5 (Peacocke, 1986) -- somewhere between the simplified top level (which Peacocke calls level 1) and the algorithm level (or level 2).

Peacocke's level 1.5 is meant to be the level of 'information on which the algorithm draws' (p. 101), but it is in fact consistent with an algorithmic level of detail in the new theory which concludes this chapter.

He seems to have an idea of an algorithm that is something akin to a program, suggesting 'a suitably constrained Augmented Transition Network, or some other favoured type of parser' as examples of algorithms. A certain amount of confusion is in evidence, however, in his support of simplifying Marr's top level, as noted above. 'The defence of this construction', he claims (p. 102 - 103),

> is that it is hard to see how finer-grained distinctions could be relevant at level 1, given that algorithms by which the function is computed are not distinguished until level 2.

But it is just that sort of finer-grained distinction that is the point of Peacocke's paper on level 1.5 and also, as shown in the chess example, of some of Marr's own discussion of level 1.

Can we pin this in-between level down any further? What exactly is this computational theory level? The answers Marr gives are largely qualitative. In my interpretation, one implication of his 1977 paper is that only some information processing problems have a computational level description, which he calls in this paper a 'Type 1 theory'. The explicitly given differences between Type 1 and Type 2 theories (both are information processing theories, however) are that the former is hierarchic and symbolic, while the latter is heterarchic and, presumably, not symbolic. The implicit differences lie in the language that is employed to describe them. Type 1 theories are referred to as being 'neatly circumscribed' (p. 39), 'clean' (p. 39 and p. 42) and 'concise' (p. 41), possessing 'beauty' (p. 41) and addressing 'deep principles' (p. 42) and 'structure' (p. 40). Type 2 theories, however, are described as 'messy' (p. 41), 'untidy' (p. 41) or 'wired-in' (p. 42), and are concerned with 'details' (p. 41) and 'performance' (p.

39) and 'mechanisms' (p. 42 and p. 45). The idea seems to be that Type 1 theories are the elegant theories, where elegance is used in the same way that mathematical elegance is used to distinguish a nice proof from a sloppy, ad hoc one. There is also the notion, again reflecting Chomsky, of having the right description at an independent level without being tied to the constraints of implementation.

Marr does emphasise, in his rhetoric at least, the primary importance of the computational level, and this can be understood in part historically. His work can be opposed to approaches to vision aimed at the level of individual neurons. At the same time, he is reacting against a style of artificial intelligence theorising in which the levels of description are confused:

> ... particular data structures, such as lists of attribute value pairs called property lists in the Lisp programming language, were held to amount to theories of representation of knowledge ... (1982, p. 28).

Although he does not necessarily live up to his own standards, Marr asserts (on p. 27) that

> it is the top level, the level of computational theory, which is critically important from an information-processing point of view.

He expands on this idea in a way that implies that by 'information-processing point of view' he means a top-down approach. An information processing problem, then, would be one such as perception (according to Marr), for which

> an algorithm is likely to be understood more readily by understanding the nature of the problem being solved than by examining the mechanism (and the hardware) in which it is embodied (p. 27).

This sounds like the sort of problem which should have a Type 1 theory, suggesting that Marr had revised his view of information processing since 1977, when both Type 1 and Type 2 theories were considered to be theories addressing information processing problems.

Disregarding the question of 'information processing', which does not seem very illuminating, let us take a look at our example in a final effort to capture Marr's computational theory level. The exclusive-or function which was put forward as the top level now appears incomplete. It can answer the 'what?' question, since there are many ways in which the exclusive-or function can be described, each of which constrains what must be computed without tying down a representation or algorithm. But there is no complementary answer to 'why?' -- why do these constraints fit the problem at hand? This is simply because we have not specified any context or wider problem. Perhaps it would be better to say that the problem specified in the example just is that of an exclusive-or calculator, reducing the 'why?' part of the question to the identity of the exclusive-or function with itself. This seems appropriate, and in fact this is what could have happened in Marr's example if he had chosen to describe a cash register as an adding machine rather than as something involved in the wider world of buying and selling.

What exactly is the computation theoretical level description for the example? I think it could be given as 'A xor B = (A or B) and (not(A and B))', where 'or', 'and', 'not' and parentheses and precedence are all taken to be previously defined. This fully describes what one usually thinks of as exclusive-or, and it would certainly be problematic if our exclusive-or calculation did not fit the constraints implied by this description.

Moreover, while it suggests a particular sort of algorithm, it is still independent of any particular algorithm or implementation in the way I think Marr means. What is left vague, insofar as Marr and Chomsky are vague on this point, is whether or why this particular formulation of the computational theory is the right one or even a good one.

Marr sees his levels approach as adding some much-needed rigour to the artificial intelligence or information processing project. While I think it is not nearly so clear and useful as he had hoped, it is certainly a step in the right direction.

NEWELL AND PYLYSHYN'S LEVELS

Like the levels of Marr, Zenon Pylyshyn's view of functional architecture, along with the theory of levels that underlies it, has been very influential and accepted on a wide scale. Particularly as it is described in Computation and Cognition (Pylyshyn, 1984), his position is often clear and frequently referenced. Consequently, I have chosen to include a more detailed look at Pylyshyn's levels, but first I want to review those of Newell, on which they are based.

Newell's Levels

Newell explicitly declares himself a realist about levels. '... computer system levels', he asserts,

> are a reflection of the nature of the physical world. They are not just a point of view that exists solely in the eye of the beholder. This reality comes from computer system levels being genuine specializations, rather than being just abstractions that can be applied uniformly. (Newell, 1982, p. 98).

In an attempt to sort out what this realism amounts to, let us look more closely at the levels Newell enumerates and the term 'specialization' (as opposed to abstraction), along with the related notion of 'technology'.

Newell's 'standard hierarchy, familiar to everyone in computer science' (p. 94; all references in this subsection are to Newell, 1982, unless otherwise noted) is given as follows (primarily taken from his Figure 2 and Table 1, both p. 96). At the bottom is the device level, which has as its medium electrons and magnetic domains. Next, moving up, is the circuit level, which has as its medium current and voltage. This seems to be something like a physical circuit, because he mentions that these are usually electric but could also be fluid. Next comes the logic circuit sublevel (also called combinatorial and sequential circuits), whose medium is single bits. Along with the next higher sublevel, the register-transfer sublevel (or architecture sublevel), whose medium is bit vectors, this makes up the logic level, whose medium in general is bits. The symbol level, which is supposed to be at the same height as the register-transfer level, is also called the program level and has as its medium symbolic expressions. The highest level (so far) is the configuration or PMS (processor-memory-switch) level, which 'lies directly above both the symbol level and the register-transfer level' (p. 95)

I think an example is desperately needed at this point. Returning to our earlier one, in which a VAX can be seen as an exclusive-or calculator, I think the whole VAX with peripherals, reading in data and writing out data, is the configuration level view. This would be more complex if there were several computers or processors; it is not just the input-output description. The program (or symbol) level is the easiest to pinpoint; it is the

PASCAL version, as in one of the possible algorithmic level descriptions of Marr. The register-transfer sublevel must be the same thing, judging from Newell's equation of these two levels, but with symbolic expressions viewed as bit vectors and variables viewed as registers or storage locations. Perhaps this could be illustrated in the exclusive-or example by the actual loaded (ready to run) object code, all in ones and zeroes with explicit addresses. Even then, the exact registers may not be known, having been left to an operating system decision to be made as the program runs. It is not clear why two levels are called sublevels or that Newell intends any profound difference between 'level' and 'sublevel'. I think they are just two ways of looking at the system in terms of its underlying logic, one bit by bit, the other vector by vector. Let us continue with the example, then. Assuming we have the object code, along with a trace of the actual registers and storage locations used (defining the logic circuit level description), then we could have the same program, but with operations broken down into operations on one bit at a time, and this would comprise the logic circuit sublevel. The plain circuit level I understand to be a physical level, as mentioned above. This would mean a description of xor in terms of the particular circuits of a VAX/750, be they Motorola 6200 chips or whatever. The device level appears to be a physical level as well, having to do with molecules and magnetic fields; it is (p. 97) 'used only to devise components at the circuit level'.

Each level must have an internal coherence; it must be a non-arbitrary system:

> Computer system levels are not simply levels of abstraction. That a system has a description at a given level does not necessarily imply it has a description at higher levels. There is

> no way to abstract from an arbitrary electronic circuit to obtain a logic-level system. There is no way to abstract from an arbitrary register-transfer system to obtain a symbol-level system. (p. 97)

Newell does not elaborate further on the implication that a specialization is different from an abstraction, but it is related to the idea of a technology, which he describes in the following way (p. 97, his emphasis):

> Computer systems levels are realized by technologies. The notion of a technology has not received the conceptual attention it deserves. But roughly, given a specification of a particular system at a level, it is possible to construct by routine means a physical system that realizes that specification. Thus, systems can be obtained to specification within limits of time and cost. It is not possible to invent arbitrarily additional computer system levels that nestle between existing levels. Potential levels do not become technologies, just by being thought up. Nature has a say in whether a technology can exist.

At the same time, Newell goes on to say that these levels are approximate and incomplete. The defining aspects of a level are (from Table 1, p. 96), with parenthetical examples from the symbol level, systems (computers), medium (symbols, expressions), components (memories, operations), composition laws (disjunction, association) and behaviour laws (sequential interpretation). In addition, the levels given share the following characteristics (also from p. 96):

> Point 1. Specification of a system at a level always determines completely a definite behavior for the system at that level (given initial and boundary conditions).

Point 2. The behavior of the total system results from the local effects of each component of the system processing the medium at its inputs to produce its outputs.

Point 3. The immense variety of behavior is obtained by system structure, i.e., by the variety of ways of assembling a small number of component types (though perhaps a large number of instances of each type).

Point 4. The medium is realized by state-like properties of matter, which remain passive until changed by the components.

Newell's main purpose in introducing and explaining the foregoing levels and concepts is to lay the foundations for proposing a new level, the knowledge level, which he claims had been conflated with the symbol level to which it remains related. The whole thing centres on the principle of rationality, which is given explicitly (on p. 102):

> If an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action.

Knowledge is defined as anything to which the principle of rationality applies, and it looks a lot like what has come to be known as 'folk psychology'. According to the defining aspects of levels, the knowledge level has as its system the agent. This is defined as bodies of knowledge, goals and actions. Its medium is knowledge, and its components are goals, actions and bodies. No composition laws are given, and it has one behaviour law: the principle of rationality (pp. 100-101).

However, the knowledge level is distinguished from its fellow levels by four 'surprises' (so-called by Newell).

These are surprises because they can be put in direct opposition to the four points listed earlier. Corresponding to Point 1, the first surprise is that specification at the knowledge level does not completely and uniquely determine behaviour at that level. In contrast to Point 2, the second surprise is that behaviour is determined globally, by the principle of rationality. The other surprises (compare Points 3 and 4) are the lack of structure at the knowledge level and the assertion that its medium is not a state-like physical structure.

Once again, we are faced with the question, what exactly is this top level? Like Marr, Newell refers to Chomsky's competence level as being what he has in mind. The principle of rationality puts some constraint on the notion. In order to extend our exclusive-or example, we must view the VAX-calculator as an agent that has as its goal the calculation of exclusive-or for some particular inputs, say, and also has knowledge of, for example, 'and' and 'or' and 'not' and how to put them into action in the appropriate manner. We still do not have any clear approach, I claim, to justify this as the 'right' knowledge level view, if indeed there is such a unique view. Newell's realism about levels seems to imply that it should be unique.

Pylyshyn's Levels

Zenon Pylyshyn (1984) takes over a sort of combination of Marr's levels and Newell's levels, and then carries the definition of the relationship between the top two levels a step further. Like Marr, he posits three levels. The bottom level is the physical level, described alternatively as biological, neurophysiological or in terms of physics. As far as I can tell, it is completely interchangeable with Marr's hardware

implementation level and is roughly equivalent to Newell's device and/or circuit levels. Similarly, Pylyshyn's symbol-processing or syntactic level fulfils the same role as Marr's representation and algorithmic level or Newell's symbol level. For instance, Pylyshyn frequently refers to this level as syntactic, using a LISP program as an example of a description of a complex system at this level, while Newell, as we have seen, also calls it the program level.

The interesting case once again is the top level, which Pylyshyn has renamed the semantic (or representational) level. The implication is that it is identical to Newell's knowledge level, as implied in the assertion '... Newell (1980, 1982) ... uses the term "knowledge level" where I use "semantic level".' (1984, p. 32), but this is not so. The name change is not important; Newell (1982) acknowledges the possible confusion in using 'knowledge' where the philosophical community would tend to use 'belief', and it is this confusion that Pylyshyn hopes to remedy by the change of terminology. The substantial difference, however, is as follows. Whereas Newell is very careful to separate the knowledge level view of a system in terms of beliefs, goals and the principle of rationality, all implemented somehow (but we do not want to say how), Pylyshyn's view relies crucially on those distinctions being reflected at the syntactic level. Beliefs and goals, at any rate, are encoded and processed to produce rational behaviour. He wants to claim (p. 40) 'that the symbols represent one content and no other' and that 'syntax parallels semantics'. The advantage to Pylyshyn of computation is that it shows us just how this can be done. He argues that the semantic level is the appropriate level for cognitive science explanations, emphasising that

> the notion of representation is necessary only in the context of explanation; it is needed to state generalizations concerning the behavior of systems under certain descriptions. (p. 26).

At the same time, the leap from Newell's view is clear when Pylyshyn asserts that

> As materialists we must ask how behavioral regularities such as those captured by statements that mention extrinsic but causally unconnected entities (or perhaps nonexistent objects) are possible in a world governed by physical laws ... The answer I ultimately give to this puzzle is that the causes of the behavior are ... some physically instantiated internal representation of such things, that is, a physical code or a symbol. (pp. 25-26, his emphasis).

Pylyshyn makes explicit what Marr and Newell seem to believe but avoid saying. That is that what makes a particular theory centred on the uppermost level right is that the theoretical entities posited at that level are real at the middle level -- at least as far as atoms and molecules and other invisible physical entities are real. The symbols or codes of the syntactic level are what parallel the semantics, while the more detailed states and operations of the syntactic level are mere messy implementations. The vocabulary continues to reflect this division as well. The highest level is also called cognitive, computational, mental and intentional, as opposed to the symbol processing 'functional architecture', which is syntactical or non-semantical, non-representational, non-cognitive and 'wired-in'. But the implication that a semantical theory should be an elegant theory is spelled out little more. The highest level semantic theories capture the right generalisations because they reflect folk psychology categories, which are intuitively plausible and have held up well through time; they are the best we have.

Apart from the reference to folk psychology, Pylyshyn gives no argument for why these three levels make up the appropriate description and explanation of complex systems. He starts from the assumption that functional psychology is a distinct and independent level from the physical, based on Fodor's irreducibility argument (Fodor, 1974; Fodor, 1975). Briefly, this is the claim that symbolic level properties and laws cannot be reduced to physical level ones, because properties at the higher level cross-classify properties at the lower level. Programming languages, for example, can be instantiated in arbitrarily many physical ways. Any attempt to characterise the behaviour of the programs at the physical level is doomed, Fodor and Pylyshyn claim. A similar argument can be made for the autonomy of the semantic level, Pylyshyn argues. In the same way, an analogy is made with the autonomy of biology with respect to physics (pp. 35-36). Like other generalisations, constraints must be stated in terms of the appropriate level. These lines of reasoning provide a persuasive case for allowing multiple levels of description, but the appeal to folk psychology and the reference to Newell, whom we have already considered, are the only support for why these particular levels are the right ones to use. Granted, they are implicit in much current theorising in cognitive psychology and artificial intelligence, but that is arguably the result of the same influence of folk psychology and arguments such as those of Fodor.

In an effort to try and understand what Pylyshyn is saying up to this point, let us take a different approach. Let us try placing our old familiar example into his framework. (He uses a similar example of binary addition on pp. 59-62 of Pylyshyn, 1984).

The states of the memory registers are designated by expressions 'x' and 'o'. Let us further suppose, following Pylyshyn closely, that when a certain pattern (designated by the symbol 'xor') occurs in a position of the machine called its 'instruction register', the machine's memory registers change states according to a specifiable regularity. So when part of the machine called 'register 1' is in the state designated by 'x' and 'register 2' is in the state designated by 'o', then 'register 3' changes into the state corresponding to 'x'. With physical and algorithmic level descriptions as before, the semantic function could be defined more in line with Pylyshyn's view as:

(1) SF[[o]] = 0
(The semantic interpretation of 'o' is the number zero.)
(2) SF[[x]] = 1
(The semantic interpretation of 'x' is the number one.)

If the computer is in the state characterised by the description:

(1) Register 1 'contains' symbol T'.
(2) Register 2 'contains' symbol T".
(3) The instruction register 'contains' the symbol 'xor'.

then the computer goes into the state characterised by:
Register 3 'contains' the string T, where
SF[[T]] = SF[[T']] exclusive-or SF[[T"]].

Although this example does not get into the 'systematicity' that Pylyshyn sees as so important, it makes clear the difference between Pylyshyn's (and Marr's and Newell's) view of the top level and that of the simplified Marr view (see earlier sections of this

chapter). This clearly says more than just the mathematical view of the exclusive-or function. In taking this definite step, Pylyshyn goes beyond Marr and Newell and into what I think Marr would call the representation and algorithm (or symbol) level. Consider Marr's idea of representations as systems, such as numeral systems; this is just what Pylyshyn is describing in his addition example. Given the difficulty in defining Marr's computational level, consider the simplified view of Marr's levels, which takes the top level as the functional extension. From this angle, Pylyshyn's semantic level is descending below the mathematical level to say something, though certainly not everything, about 'how' a function is carried out.

OTHER VIEWS OF LEVELS

There are numerous other views of levels in the literature. The ones that follow are very influential and have certainly influenced what follows in this thesis. The revised view of levels presented at the end of the chapter has grown out of at least all of these, along with the views of Marr, Newell and Pylyshyn. The exchange between McClelland and Rumelhart and Broadbent is a particularly good illustration of some of the confusion surrounding levels talk.

Dennett's Stances

A common influence on most approaches to levels, including those that are reviewed here, is likely to be found in the early work of Daniel Dennett on 'intentional systems'. Without worrying about his motivation for now (but see the final section of this chapter), let us consider his definitions of 'levels of description', or 'stances', as he calls them (Dennett, 1971, p. 221 in Mind Design).

The systems with which Dennett is concerned are complex ones, such as people or chess-playing computers, for which one can and often does 'explain and predict their behavior by ascribing beliefs and desires to them ...' (p. 225, again in Mind Design; all further references to Dennett, 1971, will be from this source). The emphasis at the top level, or 'intentional stance', here should be placed on prediction of output from input, without necessarily explaining anything about how the actual mechanisms of this particular system serve to bring about the predicted results or what states the system goes through. The middle level addresses the mechanisms; this is clarified in Dennett, 1987. A word of caution is in order, however, as this design level is often elaborated in terms of programs (see Chapter III and VI for more on the relationship between programs and algorithms.)

Exactly what are Dennett's descriptive stances? By name, from the bottom up, they are the 'physical stance', the 'design stance' and the 'intentional stance', closely parallelling the three levels of Marr and Pylyshyn. At the physical level,

> predictions are based on the actual physical state of the particular object and are worked out by applying whatever knowledge we have of the laws of nature. (p. 222).

The spirit of this level appears to be entirely analogous to that behind Marr's physical implementation level. In fact, there is widespread agreement about this level in the views presented here, the exception being Newell, possibly, who could be said to subdivide the physical level. Dennett explicitly relates his stances to Marr and Chomsky in The Intentional Stance (Dennett, 1987, pp. 74-6) -- see below.

One point emphasised in Dennett's earlier paper is that the physical stance is the appropriate vantage point from which to predict malfunctions or outcomes not designed into the system such as 'If you turn on the switch you'll get a nasty shock.' (p. 222) It is not just the application of the natural laws to a physical state, then, that places a description at the physical level; there is some ambiguity about whether a particular description, taken out of context, is from the design or physical stance. If, for example, the system in question was designed particularly to give shocks to switch-turners, then the identical prediction 'If you turn on the switch you'll get a nasty shock' would be a prediction from the design stance. However (p. 222)

> the physical stance is generally reserved for instances of breakdown, where the condition preventing normal operation is generalized and easily locatable...

It should come as no surprise that predictions from the design stance are defined as being generated by assuming each functional component will perform as designed. The design includes the arrangement of the functions to fulfil their purposes. This use of 'function' highlights the purpose-relative definition of the term, which tends to be forgotten in computer science lingo. Even so, the design stance is close to being the same as the representation and algorithm level of Marr or the symbol level of Newell and Pylyshyn, for it is of this stance that Dennett says,

> If one knows exactly how the computer is designed (including the impermanent part of its design: its program), one can predict its designed response to any move one makes by following the computation instructions of the program (p. 221).

However the design here includes the design of the particular machine, and therefore diverges from the machine independence that is the hallmark of Pylyshyn's and others' similar level.

Predictions at the highest level, from the intentional stance, assume not only (as in the design stance case) that the machine will perform as it was designed to do, but also that the design is optimal in a certain way. This begins to sound very close to Newell's knowledge level, which does not imply any particular structure, but is founded on the principle of rationality. It should also be noted that Dennett draws his examples primarily from explanation and prediction in the sense of folk psychology, such as a chess player attempting to guess the next move of his or her opponent. Parallelling the descent from the design to the physical stance in case the assumption of proper functioning is violated, when the design is less than optimal or rational, one moves from the intentional stance to the design stance for explanation.

In The Intentional Stance, especially in the third chapter, Dennett reaffirms and clarifies his position. In particular, he explicitly calls the intentional stance an idealisation, such as the highest level, the computational level, of Marr and the notion of competence of Chomsky. Unlike Marr, however, Dennett clearly allows for the possibility that a good idealisation may have little to do with the correct theory at a lower level:

> The fact about competence models that provoked my 'instrumentalism' is that the decomposition of one's competence model into parts, phases, states, steps, or whatever need shed no light at all on the decomposition of actual mechanical parts, phases, states, or steps of the system being modelled -- even when the

> competence model is, as a competence model, excellent. (pp. 75-76).

Dennett goes on to defend the intentional stance as an excellent competence model (relating it explicitly to Chomsky and Marr), still using prediction and explanation primarily in folk psychological terms. He is interested in grounding mental state talk as it is used in our common language and understanding of each other, rather than in cognitive psychology. While continuing to mention 'explanation', prediction (or weak equivalence) carries the weight, at least at the highest level. In addition, Dennett justifies the particular idealisation of the intentional stance in terms of evolution towards rationality.

Haugeland's Levels and Dimensions

In his influential paper, 'The Nature and Plausibility of Cognitivism' (Haugeland, 1978), John Haugeland addresses the nature of scientific explanation in general, before applying his approach to 'cognitivism', or computational psychology of the more traditional sort. In particular, he considers three types of explanation: derivational-nomological (a restriction of deductive-nomological), morphological and systematic. Our primary concerns here are levels and dimensions, which are defined in terms of systematic explanation. Systematic explanation is appropriate for explicating complex systems in which the behaviour of the whole crucially involves complex interaction of the parts. We can assume, for this section, that this is the right sort of explanation for many questions about classical artificial intelligence chess programs, for example, as Haugeland suggests, and possibly for computational psychology. In Haugeland's words, systematic explanation (he is describing the example of an automobile engine in particular)

> requires specification of a complexly organized pattern of interdependent interactions. The various parts of an engine do many different things, so to speak 'working together' or 'cooperating' in an organized way, to produce an effect quite unlike what any of them could do alone. (p. 247 -- page numbers refer to the version in Haugeland, 1981).

Some further explanations may require 'reduction' of the original system, perhaps explaining its components in the derivational-nomological or morphological style or as systems once again. In any case, Haugeland calls this 'systematic reduction', after the type of explanation being reduced. Particularly in the case of systems reducing to other systems, we have a systematic hierarchy, such as discussed by Simon (1969) and 'levels' of explanation. Ultimately, such a reduction will reach a level at which the explanation is of a different, non-systematic, sort. Any lower level beyond this, therefore, would not be the result of a systematic reduction.

In this context, Haugeland gives a fairly precise definition (unlike Marr) of an information processing system; it does seem to me to be very much like what Marr had in mind. It is an 'intentional black box' which can be explained (systematically) without 'de-interpreting'. This definition needs two more definitions to be understood. First, an intentional black box is, simplifying somewhat, some system toward which one can take the intentional stance, using Dennett's terminology, with some additional constraints on the structure of the inputs and outputs. The 'intentional interpretation' of inputs, outputs and systems is grounded in 'a regular general scheme' (p. 255) for determining the meaning of tokens in terms of their simple or complex types and, pivotally (giving rise

to the 'fairly' in 'fairly precise definition' above), in showing empirically

> that under the interpretations the actual outputs consistently make reasonable sense in the context (pattern) of actual prior inputs and other actual outputs. (p. 256).

Second, by 'without de-interpreting' an intentional black box, Haugeland means, in his own words,

> explaining its input/output ability in terms of how it would be characterized under the intentional interpretation regardless of whatever other descriptions might be available for the same input and output behavior. For example, if our chess player is an IPS [information processing system], that means there is a systematic explanation of how it manages to come up with legal and plausible moves as such, regardless of how it manages to press certain type bars against paper, light certain lights, or do whatever it does that gets interpreted as those moves. (pp. 258-259).

One can think of an information processing system being expanded as it is explained along the plane of intentional interpretation (see Figure 2.1).

If an explanation descends below the plane on which explanations are in terms of chess, then Haugeland uses the term 'change of dimension' (p. 263) to describe this sort of intentional reduction, since 'change of level' has already been used to describe the levels along that plane. For example, another dimension of explanation of the chess system might be in terms of linked lists (in case it were written in the LISP programming language, especially) or even electrical circuits. We get a picture such as that in Figure 2.2. These figures help to illustrate what I understand Haugeland to mean when he says

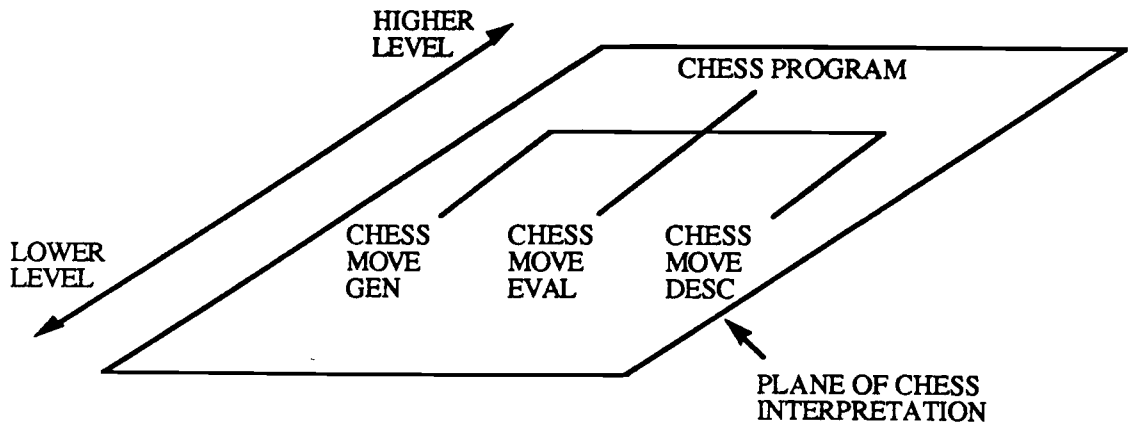
HIGHER
LEVEL
CHESS PROGRAM
LOWER
LEVEL
CHESS
MOVE
GEN
CHESS
MOVE
EVAL
CHESS
MOVE
DESC
PLANE OF CHESS
INTERPRETATION

FIGURE 2.1

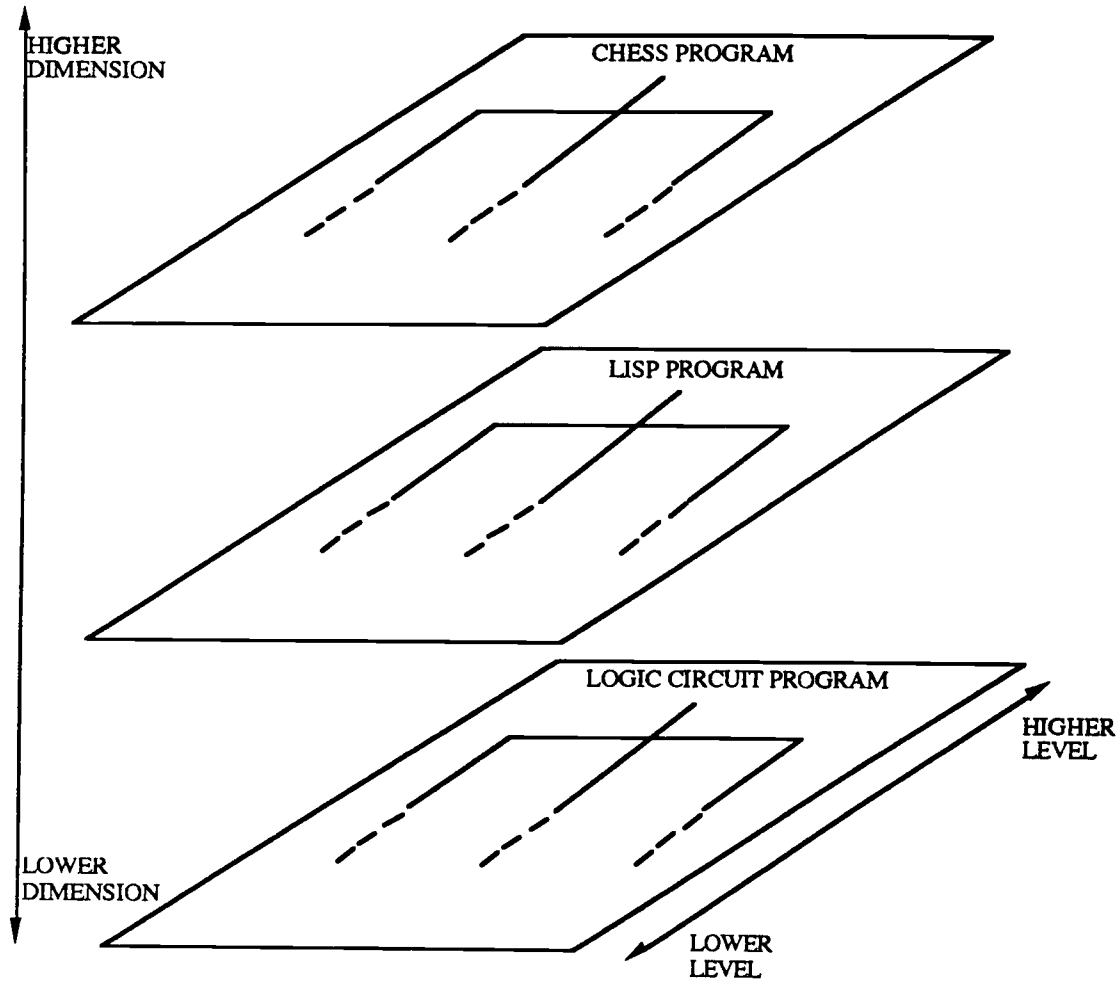
HIGHER
DIMENSION
CHESS PROGRAM
LISP PROGRAM
LOGIC CIRCUIT PROGRAM
HIGHER
LEVEL
LOWER
DIMENSION
LOWER
LEVEL

FIGURE 2.2

> There can be different level hierarchies in different dimensions, but they are 'orthogonal' rather than sequential. That is, it is a mistake to think of the lowest level on one dimension as a higher level than the highest level on a lower dimension. Thus, an and-gate is not a higher level component than a disk memory; they are components on different dimensions, and hence incomparable as to level. (p. 264).

McClelland, Rumelhart and Broadbent

A good illustration of the confusion about levels can be found in the exchange between McClelland and Rumelhart and Broadbent in 'Distributed Memory and the Representation of General and Specific Information' and its follow-up (McClelland and Rumelhart, 1985; Rumelhart and McClelland, 1985; Broadbent 1985). As a result, I shall go through it in some detail before presenting the approach to levels that is ultimately reached. In the initial article, McClelland and Rumelhart put forward a distributed, connectionist model of some aspects of memory, opposing it primarily to Morton's (1979) logogen model. The details of the data and the models do not concern us here so much as the ensuing discussion. It is there that the two authors attempt to defend themselves against Broadbent's accusation that they are presenting a theory at the wrong level, the implementation level.

In the target article, Marr's levels are not referenced explicitly, but his theory or something similar is assumed. For example, the authors acknowledge a human implementation level as well as the view that psychology in general, and their model in particular, addresses something more abstract. This can be seen when they ask themselves the following question.

> It may be conceded that distributed models describe the physiological substrate of memory better than other models, but why should we assume that they help us to characterize human information processing at a more abstract level of description? (McClelland and Rumelhart, 1985, p. 184).

They go on to answer their own question by taking two different tacks, the combination of which leads me to believe that their underlying theory of levels is very close indeed to the simplified view of Marr's theory as presented earlier.

The first and most direct response to the question is that their model is 'some of both' (p. 184), that is, it is partly a different level of description to the logogen model and partly a different description at the same level. In the simplified version of Marr's levels, this would make sense. If Morton's model defines an abstract function which makes testable predictions, then McClelland and Rumelhart's model can be seen as defining an algorithm, which is a description at a different level. At the same time, the input-output profile of that algorithm -- the model viewed at the higher level -- can be seen as a different description at the same level. It is not clear that Marr's actual view is in conflict so far; he acknowledges the intermingling and mutual constraints of all levels, though his reference to Chomsky may be at odds with this. The evidence that McClelland and Rumelhart have something closer to the simplified, as opposed to the actual, levels of Marr can be found in their discussion of the 'correct cognitive level' (p. 184), which could be the same as the computational level. They proceed to argue that their model cannot be just an implementation of logogen-like models because there are differences in the predictions made by their model and enumeration models (models in which memory traces are 'enumerated', or stored

separately, as in the logogen model). In Marr's levels, there could well be differences in prediction between computational and lower level descriptions resulting from the idealisation of the computational level. In other words, it is possible that some logogen-like model is the right computational level theory and a distributed model, such as McClelland and Rumelhart are advancing, is an implementation of it -- even if they do make different predictions. (It should also be noted that the issues are further confused by McClelland's and Rumelhart's frequent comparisons of distributed and more traditional models of memory in general. This sort of confusion will be taken up in Chapter VI.) The second strand of the response to different levels, or different descriptions, implicitly assumes that all psychological models are also algorithmic models. For example, it is pointed out that although models may share the same (behavioural) predictions, they can differ at 'a process level' (p. 185). In summary, McClelland and Rumelhart implicitly assume a view of levels close to the simplified version of Marr. Further, they assume that all psychological models are algorithmic models. Their position is clarified as they defend themselves against the criticisms of Donald Broadbent.

Broadbent (1985) attacks McClelland and Rumelhart for addressing the wrong level. While acknowledging the plausibility of distributed memory underlying the logogen theory, he invokes Marr's levels explicitly in claiming that even though logogens are part of what you would call a computational level theory, the distributed memory model is at the implementational level. Surprisingly, there is no mention of Marr's 'algorithm and representation' level, and Rumelhart and McClelland get a lot of mileage out of this omission. In fact, Broadbent's idea of levels seems to be the following. He lumps together the algorithmic and implementational

levels of Marr and calls them simply 'implementational'. For example, he states that

> M & R regard certain behavioral experiments as relevant to their model; the present point of view is that those experiments have no bearing on the question of distributed versus specific traces, but have their impact at a different level of theory. (p. 189)

It is also implied by the use of 'behavioral' in this quote that Broadbent too is taking the simplified view of the computational level as the mathematical input-output level. Further convincing evidence can be found on page 190, where he asserts that, based on the equivalence in computational power (the weak Turing machine notion of equivalence),

> ... the conclusion of the conceptual analyses of the 1950s was that the distinction of distributed versus specific representations had no importance at the computational level with which psychology deals.

Rumelhart and McClelland have brushed up on Marr's levels for their response. Adding a disclaimer (they do not necessarily agree entirely with this version of levels), they give a clear presentation of Marr's view. Whether or not they appreciate all the subtleties of the computational level is still open to interpretation:

> Casual appeals to Turing's thesis may be sufficient to establish equivalence at the computational level ... (p. 195).

Also,

> At the computational level, it does not matter whether the theory is stated as a program for a Turing Machine, as a set of axioms, or as a set of rewrite rules. It does not matter how long it takes ...

This can be contrasted to Marr's implication that in the case of computational theories for chess at least, it may indeed matter how long it takes (1977, p. 38, or see the first section of this chapter).

Their answer to Broadbent takes essentially two parts, both based on the algorithmic level. First, they rightly claim, Broadbent totally ignores this level in his criticism. (To be fair, McClelland and Rumelhart mention 'implementation' and 'predictions' and do not specifically discuss 'algorithms' in their original paper either.) Second, they disagree with Broadbent's assertion that the computational level is the right level for theories of cognitive psychology. In fact, they claim that while all levels are of interest, it is primarily the algorithmic level that is the correct province of psychology:

> It is the algorithmic level at which we are concerned with such issues as efficiency, degradation of performance under noise or other adverse conditions, whether a particular problem is hard or difficult, which problems are solved quickly, which take a long time to solve, how information is represented, and so on. (Rumelhart and McClelland 1985, p. 194).

Despite some lingering confusion over the role of the actual hardware and the status of connectionist versus classical psychology in general (see Chapter VI for more on this), I think this is just the right response to Broadbent. Furthermore, they go on to extend the standard view of levels by suggesting that the algorithmic level should itself be subdivided into many levels of analysis (p. 196). I will have more to say about this in the next section as well as in subsequent chapters.

This idea of sublevels appears in both of the 'Other Notions of Levels' (p. 195) discussed by Rumelhart and McClelland, and it is appropriate that they be reviewed here. First is the idea of programming language levels. PASCAL, for example, is a 'higher level language' than VAX assembler code. This, they claim, is what most of their critics who cite levels have in mind:

> They only believe the psychological models are more simply and easily stated in an equivalent higher level language -- so why bother? (p. 195).

The problems with this argument, they assert, are twofold. One problem is that the relationship between a PASCAL program and the assembler program obtained by compiling it is a special one; there is no equivalent theory compiler in science. The other problem is that, while there is an assembler version of every PASCAL program, this relationship does not hold in the opposite direction; most assembler programs have no equivalent in PASCAL. Note, however, that this is not clear in general. Any assembler language can be emulated by PASCAL. This is yet another illustration of the need for precision in discussing levels and strong equivalence of algorithms.

The second notion of sublevels, which McClelland and Rumelhart find much more appealing, is that of macroscopic theories and microscopic theories. Their example is Newtonian mechanics (or the logogen model) as the macroscopic theory and quantum field theory (or their own distributed model) as the microscopic theory. The two theories, in either case, make many of the same predictions, but where they diverge, it is the microscopic theory that is correct. Macroscopic theories can be seen as frequently useful approximations to the microscopic theories, the latter of which are (it

is implied) more correct. It is this view of levels that McClelland and Rumelhart like best. It can be summarised as having a number of algorithmic sublevels, bounded by the computational and implementational levels. A higher sublevel is a 'useful approximation' of a lower sublevel, as outlined above.

COMMENTS AND CONCLUSIONS: A NEW PICTURE OF LEVELS OF DESCRIPTION OF COMPLEX SYSTEMS

In cognitive psychology and in computer science the notion of levels is an important one. To speak only in terms of one level of components, such as 'memory' or 'lists' rules out the possibility of further explaining how those components are themselves structured and how their subcomponents interact. On the other hand, to limit explanation or even description to the lowest level (if that could be determined), say the level of physics, would be to miss out on important generalisations. The fundamental importance of levels is as uncontroversial as anything in cognitive science. In support of this claim, it is one of the few initial points of agreement between McClelland and Rumelhart and Broadbent in their exchange described in 'Other Views of Levels'. Still, the variety of levels, as we have seen from a few representative viewpoints, is wide-ranging, and the definitions are less than precise. With some minor modifications, this situation can be put right, laying the groundwork for more precision and clarity in the related concepts of algorithms and abstraction, topics which will be the focus of the next chapters.

What Are Levels Levels of?

My view is that levels are not 'real' in the way that Newell suggests (1982, discussed earlier in this

chapter), but rather they are levels of description, especially levels of description of complex systems and often for the purpose of explanation. This is a weak view: support for this minimal claim comes down to nothing more than the importance of levels -- they are at least levels of description. Brian Smith (1986, p. 42) states the difficulty clearly:

> I don't understand what it is to say, for example, that a level 'really exists', rather than being a point of view. Usually there are theoretical frameworks under which we describe computational devices; in one sense these are viewpoints, but that fact doesn't make the objects viewed from those viewpoints any less real, or descriptions in the viewpoints' terms any less true.

If there is a claim that levels are more than descriptions or points of view, then an argument is necessary. Newell in fact claims more, at least for levels of computer systems, but his attempts to support the claim are insufficient. They hinge on the levels as systems realised in 'technologies'. A technology enables the realisation of levels in a routine manner in Newell's description, allowing for estimates of time and cost. In this sense, there is a 'circuit technology' and a 'programming language technology' (although it should be noted that development times and costs of computer systems, especially software, are notoriously difficult to predict). Let us look more closely at the programming language level or symbol level. Given the wide range of programming languages, a chess-playing computer, for example, might be described at the level of machine language, assembler language, LISP, or a high level language built on top of LISP. Between any two of these sublevels, an interim programming language could be devised. Granted, these are sublevels within the 'single' symbol level, at least at first glance. On closer inspection, there is no clear boundary between the

symbolic level and the logic level. A description in machine language, for example, might be viewed as either. Furthermore, the appropriate description of a system in some context may cross these technological boundaries, giving a high level perspective as background for a more detailed, low level description of a particular function. The crucial claim that each level must itself be a system is not supported separately. My position on this issue (developed in the next chapter) is that some levels of abstraction may be conveniently characterised as systems in the way I think Newell intends, but this is not an essential requirement for defining a level.

Perhaps what Newell has in mind as levels are more like the dimensions of Haugeland (1978, also discussed in this chapter in 'Other Views of Levels'). It is the subject matter that changes between dimensions, from linked lists to bit vectors in the case of the symbolic dimension to the logic dimension. In the case of computers it is tempting to think in terms of only these levels, given that there is usually an application, a programming language and a particular machine in question. As a result, the technology argument may be more appropriate for systems which have been engineered in terms of particular technologies as opposed to the analysis of 'systems' such as people, whose design history is not entirely clear. But even an engineered system may be described in terms other than those of its underlying technology if it is seen as a system other than the one it was designed to be.

How Many Levels?

Assuming our levels are levels of description, how many should there be? The answer I shall advocate is 'It depends', or 'Arbitrarily many, depending on the context and the sort of description or explanation required'.

However, there have been several exponents of a particular number of distinguished levels, notably three levels. The arguments in the section above are related, since Newell claimed in defence of the reality of levels that new levels could not be sandwiched arbitrarily in between existing levels. He proposed five levels or perhaps six levels, depending on whether sublevels are counted as 'real' levels or not. Just as Newell's reasoning for his particular levels was weakened by removing constraints from the definition of levels, so are the cases for the more common starting point of three primary levels. We have seen three advocates of various versions of this starting point; let us consider each of them in turn.

To begin with, Marr (1982) suggests three levels without specifically arguing that there should be no more. As has been noted, the top level at times appears algorithmic. While Marr's levels, or the simplified version of them, have been the accepted foundation of level talk, recently they have attracted criticism, especially regarding the individuation of three levels. Sejnowski, Koch and Churchland (1988), for example, take what counts as computational or implementation to be dependent on where you are in the picture (p. 1305):

> ... a model of an intermediate level of organization will necessarily simplify with respect to the structural properties of lower level elements, though it ought to try to incorporate as many of that level's functional properties as actually figure in the higher level's computational tasks. Thus, a model of a large network of neurons will necessarily simplify the molecular level within a single neuron.

Similarly, Lycan (1987) points out that three levels are not enough, even in the better understood context of computers. We have already seen an argument by

McClelland and Rumelhart that the algorithmic level should be divided into sublevels. The main force of all of these persuasive counterexamples is as follows. First, it is difficult to separate the proposed three levels clearly. Second, it is easy to create levels in between the given levels. Third, point of view and context cannot be ignored. Interestingly, in an early paper (Marr and Poggio, 1977) even Marr advocated four levels, splitting the physical level into two parts (p. 470):

> For a system that solves an information-processing problem, we may distinguish four important levels of description. At the lowest, there is basic component and circuit analysis -- how do transistors, neurons, diodes, and synapses work? The second level is that of particular mechanisms: adders, multipliers, and memories accessed by address or by content. The third level is that of the algorithm, and the top level contains the theory of the overall computation.

Pylyshyn's arguments for three levels are based, he claims, on Newell's levels, but Newell himself actually advocated five or six levels, at least for computer systems. Beyond that, Pylyshyn presents a good case for a number of different levels capturing the appropriate generalisations, yet says nothing that justifies limiting that number to the three he has in mind. Pylyshyn does say there are 'at least' three levels and at times seems to divide the physical level into separate physics and biology levels. Levels are justified by their explanatory power, their ability to capture useful generalisations. While agreeing that different levels are required to capture different generalisations, I claim that the appropriate level of description or explanation cannot be determined without a particular question in a particular context.

The design and physical stances of Dennett could (and I think should) be applied to a variety of levels of algorithms. From the design stance, only those aspects that were intentionally designed into an algorithm are considered. The physical stance covers the remaining assumptions that were not explicitly part of the design. The intentional stance is represented as

> a rationalistic calculus of interpretation and prediction -- an idealizing, abstract, instrumentalistic interpretation method that has evolved because it works and works because we have evolved. (Dennett, 1987, pp 48-49).

It is what we resort to when attempting to interact in a common sense way with complex systems such as chess-playing computers or other human beings. Dennett's intentional system theory is to justify intentional talk, talk about beliefs and desires, and its usefulness in everyday life. His arguments for this level do not carry over so well to scientific explanation in psychology, nor are they meant to do so.

In conclusion, there is no clear cut case for limiting the number of levels for the purposes of cognitive psychology, while there is good reason to allow many levels based on the context and purpose of description or explanation. In accord with Dennett, but with an eye to more levels, I seek a formal and systematic foundation in part based on Turing's foundation for computability theory.

What is the Relationship between Levels?

With the exception of Haugeland's, all the theories of levels discussed so far have a common characteristic. They all place the proposed levels as variations along a single dimension, from less to more abstract. This

applies to the more recent multilevel suggestions, as well, which I think represent movement in the right direction. They add extra steps along the same dimension. In addition to retaining that single dimension, they do not question the idea of a high mathematical or computational level and a low physical level, though it is accommodated in different ways. McClelland and Rumelhart, to start with, place the computational level and the physical level at (positive) infinity and negative infinity, if you will, along the dimension. They are boundary levels, but they are accessible in a way that infinity may not be -- I do not care to push this number line analogy too far. In between, of primary interest to cognitive psychology, are the many algorithmic levels. Sejnowski, Koch and Churchland, on the other hand, put the computational and implementational levels at one above and one below the current algorithmic level, labelling levels only relatively. Churchland and Sejnowski (1988) also make the important point that all formally described levels are independent of the lower levels in that they can be implemented in many ways (p.742):

> Computational theory tells us that algorithms can be run on different machines and in that sense, and that sense alone, the algorithm is independent of the implementation. The formal point is straightforward: since an algorithm is formal, no specific physical parameters (for example, vacuum tubes or Ca^{2+}) are part of the algorithm.

More will be said about this in the next chapter.

Returning to the old exclusive-or example, we might have the following views of levels. Let us start with the simplified Marr version. As shown in Figure 2.3, this is just the basic three levels. The computational level

```
COMPUTATIONAL LEVEL (TOP)

XOR:   (0,0) --> 0; (0,1) --> 1; (1,0) --> 1;
       (1,1) --> 0
```

```
ALGORITHMIC LEVEL

get x;  get y;  if x = y then z = 0,
                otherwise z = 1
                output(z); end
```

```
IMPLEMENTATION LEVEL (BOTTOM)

L15: cmpl 4(ap), 8(ap); jneq L16;
clrl -4(fp); jbr L17; L16: movl $1,-4(fp);
L17: ...
```

FIGURE 2.3

is represented in the figure by the exclusive-or mapping. This is meant to be the abstract mathematical function, saying nothing about how such a function might be carried out. At the bottom, some of the VAX assembler version has been used to represent the hardware or implementation level. In between is a possible algorithm connecting the two.

That there are in fact levels between these is illustrated in Figure 2.4. Now, on either side of the old algorithmic level, between it and the outer levels are more (shall we say algorithmic?) levels. Rumelhart and McClelland would call these subdivisions of the algorithmic level, while Sejnowski et. al. would say that either of the three middle levels could be considered algorithmic, and the levels above and below each one could be considered computational and implementational, respectively.

Furthermore, why should we be content to add levels only between the original three? At the bottom, we could go beyond the level of the VAX assembler (in Figure 2.5, an attempt is made to represent this lower level with machine level binary coding), even down to the level of molecular interaction. At the top, we could abstract from the two-valued function to a nondeterministic one-valued relation, by considering only the first input. This would give the relation taking 1 to 0, 0 to 0, 1 to 1 and 0 to 1, as is also illustrated in Figure 2.5.

I am, as stated earlier, entirely in agreement with a many-layered theory of levels of description. At the same time, there is still something worrying about the picture in Figure 2.5. Each line in the figure could be interpreted as an algorithm, but each one could also be taken to be a formal, mathematical object in which only the extension is significant. This is how algorithms

COMPUTATIONAL LEVEL (TOP)

XOR: (0,0) --> 0; (0,1) -->1; (1,0) --> 1;
(1,1) --> 0

---

get 2 inputs; use equality check:
x = y => 0; x not = y => 1

ALGORITHMIC LEVEL

```
get x;  get y;  if x = y then z = 0,
                otherwise z = 1;
                output(z); end
```

```
get x;  get y;  if x not in {0,1} or y in {0,1} then
stop, otherwise
                if x = y then z = 0,
                otherwise z = 1;
                output(z); end
```

IMPLEMENTATION LEVEL (BOTTOM)

```
L15:  cmpl 4(ap), 8(ap);  jneq L16;
clrl -4(fp); jbr L17;  L16: movl $1,-4(fp);
L17: ...
```

FIGURE 2.4

```
0 --> 0;  0 --> 1;  1 --> 1;  1 --> 0
```

```
COMPUTATIONAL_LEVEL (TOP)

XOR:  (0,0 --> 0;  (0,1) --> 1;  (1,0) --> 1;
       (1,1) --> 0
```

```
get 2 inputs;  use equality check:
               x = y => 0;  x not = y => 1
```

```
ALGORITHMIC LEVEL

get x;  get y;  if x = y then z = 0,
                otherwise z = 1;
                output(z);  end
```

```
get x;  get y;  if x not in {0,1} or y in {0,1} then
stop, otherwise
                if x = y then z = 0,
                otherwise z = 1;
                output(z);  end
```

```
IMPLEMENTATION LEVEL (BOTTOM)

L15:  cmpl 4(ap), 8(ap); jneq L16;
clrl -4(fp); jbr L17;  L16: movl $1,-4(fp);
L17: ...
```

```
00111101110000101001l ...
```

FIGURE 2.5

tend to be viewed in proofs of equivalence or correctness, for example. Similarly, each line could be implemented in a physical device. The algorithm would then describe, more or less completely, the workings of that device, relative to a certain perspective. In other words, there is something very appealing in the three (simplified) levels of Marr, and yet the near-continuum of algorithms along a dimension (as might be developed along the lines of Figure 2.5) should not be thrown away.

In Figure 2.6, I have tried to retain, but at the same time to separate, at least two senses of 'abstraction' whose conflation has contributed to the confusion of levels. Along one dimension, top to bottom, is abstraction in terms of concreteness. These levels are close to those of the simplified Marr view, and they are represented by planes in the figure. The second sort of abstraction has to do with the amount of detail. On the algorithmic plane, therefore, I have indicated some of the continuum of levels from the earlier figures. In terms of this picture, we can take 'abstraction' to mean any of three types of moves. It can mean a move from one plane up to a higher plane, or it can be a move from one level on a particular plane to a less detailed level on the same plane. In addition, it can be a combined move to less detail on a plane and then up to a higher plane (or up to a higher plane first, and then to less detail on that plane). There need not be just a single path in any of these directions. From a particular starting point, such as 'get x; get y; etc.', abstraction in either of the two basic senses can take any of a large number of possible directions. Similarly, details of an algorithm can be added in many different ways, each of which can be implemented in many sorts of hardware. I will try to reserve 'implementation' primarily for implementation with respect to detail and use 'realisation' for implementation in a concrete physical

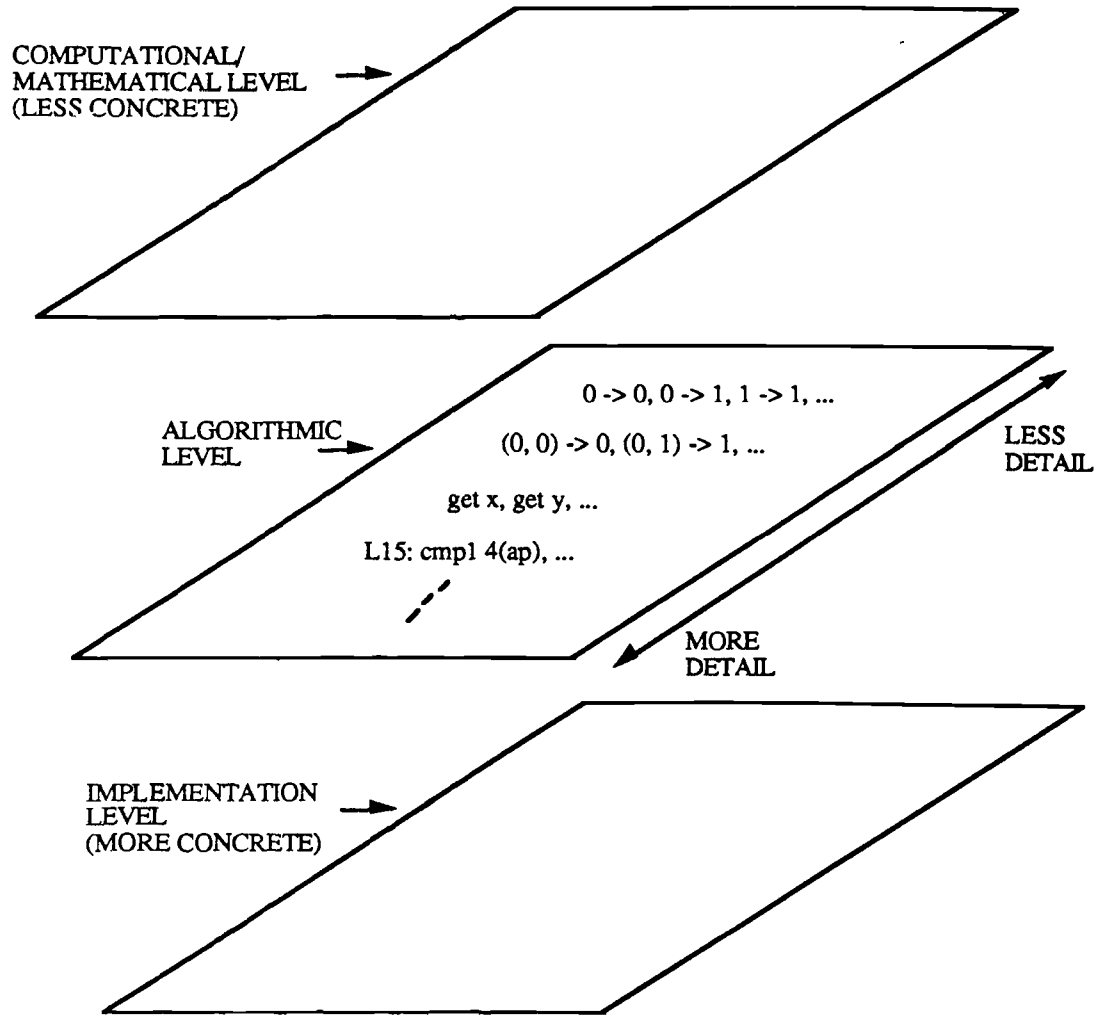
COMPUTATIONAL/
MATHEMATICAL LEVEL
(LESS CONCRETE)
ALGORITHMIC
LEVEL
0 -> 0, 0 -> 1, 1 -> 1, ...
(0, 0) -> 0, (0, 1) -> 1, ...
get x, get y, ...
L15: cmpl 4(ap), ...
LESS
DETAIL
MORE
DETAIL
IMPLEMENTATION
LEVEL
(MORE CONCRETE)

FIGURE 2.6

device. The possible interrelationships of levels and the clarification of 'algorithm' and 'abstraction' will be further explored in the next chapter.

It may be recalled that Haugeland was the exception to the earlier views of levels along a single dimension. His two-dimensional view (my use of dimension should not be confused with Haugeland's use of the word to label one particular direction) no doubt influenced my two-dimensional formulation indirectly, but it is somewhat different. While both versions share the idea of a change in level as a variation in the amount of detail, they diverge in what is shown as a change in the other direction. This 'other direction' is referred to by Haugeland as a change in dimension, and it can be seen as a change in subject matter, from chess to lists, for example (see Figure 2.2). The concreteness aspect of abstraction is not so apparent: Chess talk might be viewed as less concrete than list talk to some, but this is debatable. Further, Haugeland takes pains to point out that a level on one dimension cannot be compared (said to be higher or lower than) a level on another dimension. In the view presented in Figure 2.6 above, on the other hand, there is a corresponding mathematical level and concrete realisation for each individual algorithmic level, so a means of comparison across levels is possible, for some algorithms at least. Haugeland's change in subject is important to capture as well, and I think it can be handled, but that is left for Chapter III.

Before going on to the important concepts of algorithms and abstraction, terms which have been used rather loosely up to this point, let us examine the new picture of levels for the sake of clarity. This should provide a means of checking that some of the pitfalls of earlier

definitions of levels are avoided, as well as giving a firm basis from which to proceed.

One way to describe the theory of levels I am advocating is to take an overview of the structure. At first glance it has a lot in common with the simplified Marr levels. There are three levels of concreteness. The highest, or least concrete level, is the level of mathematical functions or, to be more general, it would be better to say mathematical relations. These may be written in any number of ways. (((0,0), 0), ((0,1), 1), ((1,0), 0), ((1,1), 0)) or 'exclusive-or' or 'Z := (x|y) & not (x & y)' are all possibilities for the usual example, because what is important is the mathematical relationship between input and output, not the representation and not how one might get from the input to the output. This level is independent of space and time, dealing as it does with abstract, mathematical relationships. It can be contrasted to the concrete physical level and the relative space and time of the algorithmic level.

Jumping down to the bottom level, we get to the concrete world of both absolute space and absolute time, the physical level. At this level we have real objects, such as the VAX which may be described at the highest level as calculating the mathematical function exclusive-or, among its many and varied possible descriptions. There is not really anything called a 'physical description', only physical objects themselves. Any description of them will be abstracted from some aspects of their physical extension. In contrast, the mathematical and algorithmic levels of description are, as the name implies, descriptive. There is no sense of more or less detailed physical systems.

The middle level, or algorithmic level, is in between the other two in terms of concreteness. It may be said to deal with the space and time taken up by mathematical functions realised in physical systems, but only at a level removed from the physical world. For example, a von Neumann algorithm to put a number of items in order can be said to operate in linear time if the number of operations it performs is proportional to the number of input items. It might also be said to require a certain amount of storage, which can again be dependent on the number of input items. These time and space measurements are abstractions, however, and do not give the exact amount of time or space involved in any particular realisation, although they provide a measure that can be used across certain classes of realisations (such as von Neumann machines) with suitable caution. The idea of algorithm, as noted, is left vague here, but it is intended to include algorithms and representations without distinction (insofar as it is possible) between process and data. Intuitions regarding algorithms, if they have not caused major difficulties so far, should suffice for now. A LISP program or the various symbol levels would be one way of viewing algorithms until we get to Chapter III.

On the algorithmic plane by itself, different levels of abstraction are distinguished in terms of detail rather than in terms of concreteness. In Figure 2.6, then, a detailed description of VAX assembler code adds more detail to an explanation of how a program in a higher level language calculates exclusive-or. The linear path from less to more detail in the illustration may be misleading; detail can be added in many possible ways.

Another way of looking at the proposed theory of levels is from the vantage point of a particular algorithm. In Figure 2.7, the white dot is meant to represent the

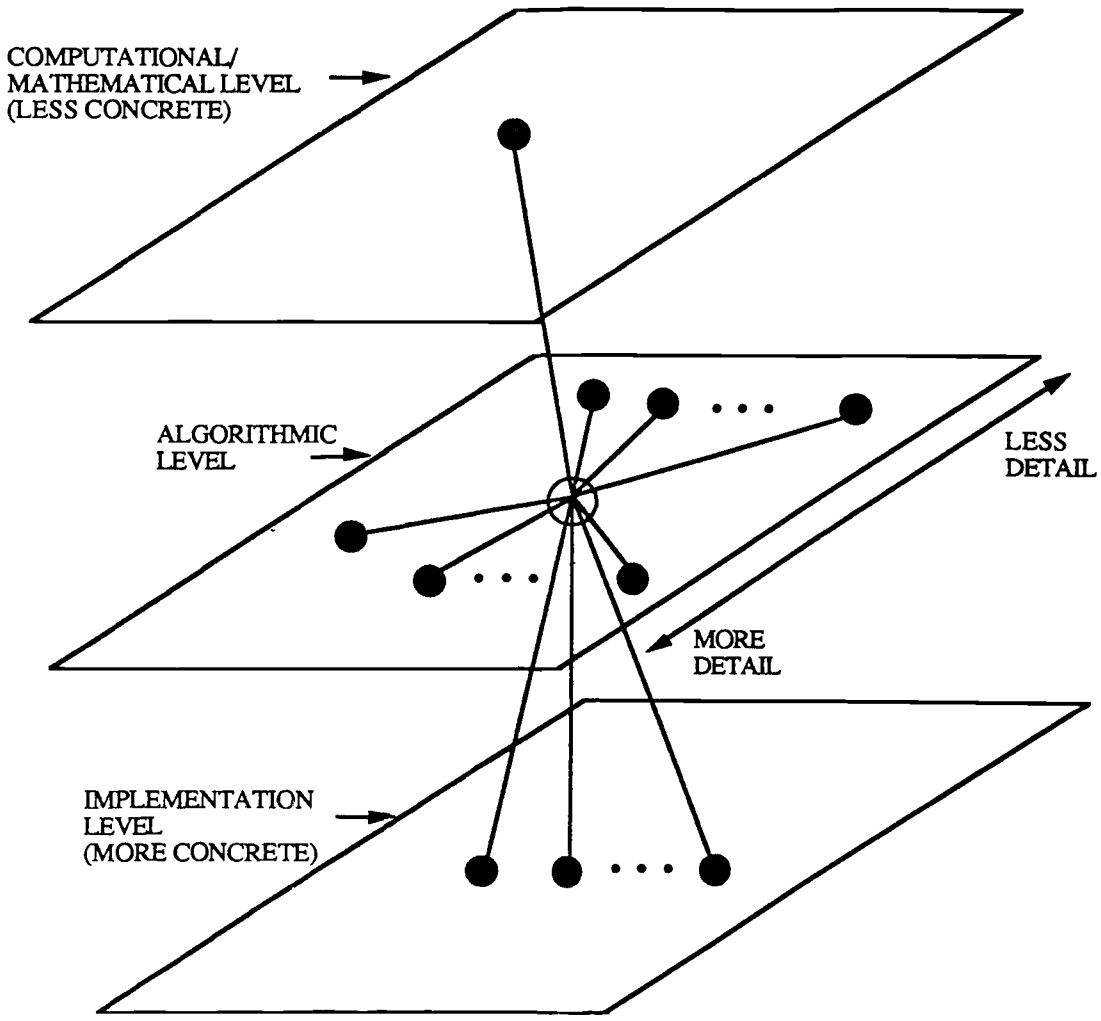
COMPUTATIONAL/
MATHEMATICAL LEVEL
(LESS CONCRETE)
ALGORITHMIC
LEVEL
LESS
DETAIL
MORE
DETAIL
IMPLEMENTATION
LEVEL
(MORE CONCRETE)

FIGURE 2.7

algorithm in question. It can be seen as a mathematical relation, by moving upward to a more abstract (with respect to concreteness) plane. Alternatively, it can be realised in arbitrarily many ways by moving down to the physical plane. On the algorithmic plane, it can also be further defined or explained with the addition of detail in an arbitrarily large number of ways. It can be a starting point for abstraction, as well, giving less specific algorithms on a higher level, a level of less detail.

With figures such as Figures 2.5 and 2.6, there is the danger that Rumelhart and McClelland (1985) warned against, the danger of incorporating the peculiar relationships between levels of programs into more general levels of description and explanation of complex systems. These problems will be addressed in the process of developing a more precise definition of algorithms, a process which includes a closer look at the relationship between software and hardware.

CHAPTER III
ALGORITHMS

THE IMPORTANCE OF ALGORITHMS

In Computation and Cognition (Pylyshyn, 1984), Zenon Pylyshyn puts forward his idea of algorithms. He writes eloquently on behalf of the importance of getting clear about algorithms and the importance of using computational models in psychology in a principled way. In fact the entire book can be taken as an argument supporting these two points. He asserts, as an example, on page 85 that

> Psychologists pursue the goal of explanation, which means that, although we pursue the constructivist program, we must make sure our use of computer models is principled. This, in turn, means that stringent constraints must be applied to the theory construction task to ensure that the principles are separated from the ad hoc tailoring of the systems to make them fit the data.

At the same time, he notes

> ... little progress has been made in formally defining the algorithmic equivalence of programs. No one has yet produced a natural, uniform way of distinguishing essential from nonessential properties in programs that applies over a variety of domains and languages. (p. 90)

In the terms of the new picture of levels put forward in the last chapter, complex physical systems such as people and machines can be described and explained at many different algorithmic and mathematical levels. Cognitive psychologists are potentially interested in any of these levels. The same is true of any other 'reverse engineering' enterprise. Imagine the task of a repairer of software who must determine the organisation of a

program written by someone else in order to fix or update it. It may be helpful to know the original purpose of the program. To take a geometric example, perhaps at the mathematical level the program can be described as calculating the direction from one point in space to another point in space, given two points in a plane as input. The various levels of description required beyond this are dependent on the symptoms of the problem (maybe the program fails if two identical points are given as input), the requirements for the change (the function might have to be extended to handle points in three or more dimensions, rather than just a plane) and on the internal structure of the program itself. If the problem seems to be that the program does not properly respond in the case of two identical input points, the algorithm may not have to be understood as thoroughly (or rather, at such a detailed level) if this case is treated separately at the beginning of the program.

So far, the term 'algorithm' has been used practically synonymously with 'program', based on one way of using it in computer science, but mainly as a device to get off the ground. The new ways of talking about levels and algorithms that I am advocating are interdependent, so some simplifications are required in order to begin at all.

This traditional way of using 'algorithm' is perfectly adequate for the modifier of software and for many other purposes. Precise definitions of this type are given in the next section. If, however, the term is used in psychology to support mental state talk, more care is needed. Much of cognitive science and cognitive psychology in recent years has been based on an analogy with computer systems (without saying which came first). In attempts to explicate the mental as opposed to the physical in the face of prevailing materialism, the

distinction between software and hardware has been a tempting toehold.

Because the computer system case is better understood and more ethical to poke and prod, and also because the languages of computer science and of cognitive science are so intertwined, there is a definite computational bias in the approach adopted here. At the same time, the motivation comes from trying to make sense of talk about mental states in cognitive science. In computational terms, that could be translated into trying to make sense of 'software states'.

What could a software state be? A first response might be the program line or statement that is currently being executed. But which program, the high level LISP version or the compiled assembler version, to take but two possibilities? What if the original LISP code is lost forever? Does the computer still move through 'software states' defined by it? What if the same steps are hardwired (built into the hardware) of another machine modelled on the first, so that it arguably goes through the same steps, but there are no programmed instructions located in a program memory?

In 'Minds and Machines' (Putnam, 1960), Hilary Putnam sets out an analogy

> between logical states of a Turing machine and mental states of a human being, on the one hand, and structural states of a Turing machine and physical states of a human being, on the other (p. 373 in Putnam, 1975),

whereby rational thought might be understood in part by understanding the '"program" which determines which states follow which.' In fact at various times, he

argued for, in his own words (1973, p. 210 in the version in Haugeland, 1981),

> the hypothesis that (1) a whole human being is a Turing machine, and (2) that psychological states of a human being are Turing machine states or disjunctions of Turing machine states.

He reached the conclusion that 'psychological states are not machine states nor are they disjunctions of machine states.' (p. 216). Rather than looking at the proper relationship of various levels of abstraction, he moves toward abandoning algorithmic psychology altogether, based on arguments for the autonomy of psychology much like those presented later by Pylyshyn (1984), among others.

The answer that I shall be advocating is that there are only states of physical machines, but that these can be described at many levels (algorithmic and mathematical). Algorithms are defined in terms of the states (also at many levels) through which they pass. In some cases these algorithmic descriptions can be abbreviated by program descriptions. Rather than a sharp distinction between physical states and mental states, there is a near-continuum of system state descriptions. At the more detailed end of the spectrum, these look more like what are usually called physical states. At the less detailed end, they are more like mental states. Higher level descriptions are more or less correct, depending on how they are aligned with lower level, more detailed descriptions. In this framework, the relation between software and hardware can be made clear. Software, a LISP program say, is a design or possibly a theory, depending on one's point of view, about how a physical system carries out a function. More accurately, at the abstract mathematical level it is a predictive design or

theory about what the system carries out. It may or may not be an algorithmic theory, specifying something about how the final state is reached, i.e. about what states are passed through in the interim. Whether software is an abbreviation of an algorithm of an actual system or not depends on the complexity of the particular compiler, the nature of the machine and the level of algorithm in question.

This is merely an introduction to provide a flavour and a context for the following discussion of algorithms. It will be clarified and expanded in what follows. A consideration of some implications of this approach is taken up in Chapters V and VI.

DEFINITIONS OF ALGORITHMS

The definition of an algorithm is typically given as a finite sequence of instructions that when followed will terminate after a finite number of steps. In addition to the requirement of finiteness, in both of these ways, there is a requirement of definiteness. The instructions must be stated precisely, even formally. Davis (1958, p. xv) is

> concerned with the problem of the existence of algorithms or effective computational procedures for solving various problems. What we have in mind are sets of instructions that provide mechanical procedures by which the answer to any one of a class of questions can be obtained. Such instructions are to be conceived of as requiring no 'creative' thought in their execution. In principle, it is always possible to construct a machine for carrying out such a set of instructions or to prepare a program by means of which a given large-scale digital computer will be enabled to carry them out.

While algorithms defined in this way may not be tied to a particular language, in a discussion of Church's thesis in Davis and Weyuker (1983, p. 54), the following explicit statement is made (with emphasis added):

> But, since the word 'algorithm' has no general definition separated from a particular language, Church's thesis cannot be proved as a mathematical theorem.

More or less precise versions of this type of definition are given to beginning computer science students. Brookshear (1988, an introductory textbook), for example, refers informally to an algorithm as a 'step-by-step process' and then states (p. 130) 'Technically speaking, computer science defines the term algorithm as a finite sequence of unambiguous, executable steps that will ultimately terminate if followed.'

On the other hand, Enderton (1972), who is concerned with as precise and mathematically correct formulation as possible, settles for something less than a particular formal language to pin down his definition of algorithms or, as he calls them, effective procedures. He gives the issue a more thorough treatment than most, the bulk of which is included in the following (p. 60):

> By an effective procedure we mean one meeting the following conditions:
>
> 1. There must be exact instructions, finitely long, explaining how to execute the procedure. These instructions should demand no cleverness on the part of the person (or machine) following them. The idea is that your secretary (who knows no mathematics) or your computing machine (which does not think at all) should be able to execute the procedure by mechanically following the instructions.
>
> 2. The procedure must avoid random devices (such as the flipping of a coin), or any

> such device which can, in practice, only be approximated.

On page 61, he continues,

> Of course the above description can hardly be considered a precise definition of the word 'effective'. And, in fact, that word will be used only in an informal intuitive way throughout this book ... But as long as we restrict ourselves to positive assertions that there does exist an effective procedure of a certain sort, the intuitive approach suffices. We simply display the procedure, show that it works, and people will agree that it is effective. (But this relies on the empirical fact that procedures which appear effective to one mathematician also appear so to others.)

While no particular language is invoked by this definition, any effective procedure must be written down somehow. For Enderton, this can be mathematical English as in the following example of a step he uses in an algorithmic version of a proof of the unique readability theorem for propositional logic (p. 41):

> 1. If all minimal vertices have sentence symbols, then the procedure is completed. Otherwise, select a minimum vertex which has an expression which is not a sentence symbol.

It is acceptable because it is underwritten by Turing machine computability, a subject to which I shall return. Turing wanted to avoid just such informal natural language descriptions.

It is somewhat surprising that some computer science textbooks (e.g., Tremblay and Sorenson, 1984, or Wirth, 1976) do not explicitly define the term 'algorithm' at all, but leap in with examples and contextual usage. On the other hand, given their goals of teaching the construction of programs in particular languages or

particular sorts of languages, and given that people do not learn best by being given only a set of formal definitions, the lack of explicit definitions may be understandable. The most precise definitions in basic texts, as mentioned above, emphasise finiteness and definiteness. Since these all seem to draw on the detailed formulation of Knuth (1973, pp. 4-6), I include his definition here, at the risk of belabouring the point. Notice that he adds items about input and output and distinguishes effectiveness from definiteness. (Ellipses are used to omit some references to the example of Euclid's algorithm, called Algorithm E by Knuth).

> 1) Finiteness. An algorithm must always terminate after a finite number of steps ... (A procedure which has all the characteristics of an algorithm except that it possibly lacks finiteness may be called a 'computational method'...)
>
> 2) Definiteness. Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case. The algorithms of this book will hopefully meet this criterion, but since they are specified in the English language, there is a possibility the reader might not understand exactly what the author intended. To get around this difficulty, formally defined programming languages or computer languages are designed for specifying algorithms, in which every statement has a very definite meaning ... An expression of a computational method in a computer language is called a program ...
>
> 3) Input. An algorithm has zero or more inputs, i.e., quantities which have a specified relation to the outputs...
>
> 4) Output. An algorithm has one or more outputs, i.e., quantities which have a specified relation to the inputs...
>
> 5) Effectiveness. An algorithm is also generally expected to be effective. This means that all of the operations to be

> performed in the algorithms must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using a pencil and paper. Algorithm E uses only the operations of dividing one positive integer by another, testing if an integer is zero, and setting the value of one variable equal to the value of another. These operations are effective, because integers can be represented on paper in a finite manner and there is at least one method (the 'division algorithm') for dividing one by another. But the same operations would not be effective if the values involved were arbitrary real numbers specified by infinite decimal expansion, nor if the values were the lengths of physical line segments, which cannot be specified exactly...

All this leaves us with the impression that algorithms are a lot like programs, being finite and in some executable language, if not a particular language. On the other hand, not all programs are algorithms, because a program may not halt; it may consist of instructions to loop forever, or even one such instruction, such as

```
100 GO TO 100
```

or

```
DO WHILE (TRUE);
```

In practice, however, the term 'algorithm' is used more loosely, referring for example to 'the heap sort algorithm', without specifying a language but implicitly making some assumptions about the language or structure of the underlying architecture of the intended real or ideal machine, described at a certain level. These assumptions will be examined more closely in the next section, 'Comparison of Algorithms'. This leads to several questions if one is trying to be clear about algorithms. Are, for example, two algorithms that implement the heap sort in two different languages the

same algorithm or different ones? In the terminology of Chapter II, I propose that they are likely to be different at the level of the programming languages, but they should be the same at a higher, less detailed algorithmic level.

While claiming that he uses the term 'in approximately its standard computer-science sense' (1984, p. 88, note 1), Pylyshyn puts forward a rather different definition of algorithm than those given above. In particular, he removes the algorithm from association with a language or a program. I would not argue that it is very far from the less formal way in which the term is used in computer science, such as in the example of 'heap sort', but I am not aware of any attempt to define this sense very precisely. Niklaus Wirth (1976) comes the closest of the authors whose computer science texts I have surveyed, stating (on p. xii) that

> Programs, after all, are concrete formulations of abstract algorithms based on particular representations and structures of data.

Later (p. xiv), however, he goes on to say that

> Although the mere presentation of an algorithm's principle and its mathematical analysis may be stimulating and challenging to the academic mind, it seems dishonest to the engineering practitioner. I have therefore strictly adhered to the rule of presenting the final programs in a language in which they can actually be run on a computer,

recognising that 'the devil hides in the details' (same page).

Another aspect of defining algorithms that must be addressed for the sake of clarity and completeness shows up in the title of Wirth's book: Algorithms + Data

Structures = Programs. So far, especially as they have been defined in terms of programming languages, no distinction has been made between algorithms and data structures, or, as is implied, between process and representation. Despite the succinct message of the title, Wirth (pp. xi-xiii) recognises that things are not so simple. Referring to Hoare (1972), he remarks that this contribution

> made clear that decisions about structuring data cannot be made without knowledge of the algorithm applied to the data and that, vice versa, the structure and choice of algorithms often strongly depend on the structure of the underlying data. In short, the subjects of program composition and data structures are inseparably intertwined.

At the same time, he defends his emphasis on data structures with the 'intuitive feeling' that objects come first, along with a pragmatic assumption that his readers have some programming experience.

Finally, here is how Pylyshyn defines algorithms. In the footnote cited earlier, he rightly points out that he is using the term in a more general sense than that of Newell and Simon (1972), in which they oppose it to 'heuristic'. We can consider algorithms to be finite and definite (though he does not use these exact words, but rather 'complete and deterministic'), even if they are carrying out a heuristic, or incomplete rule-of-thumb approach to solving a problem. He goes on, however, to outline a picture that contrasts algorithms as types to programs as tokens:

> Thus we can have different programs for the same algorithm (one might be in FORTRAN, another in PASCAL, and still another in LISP). Because algorithm is a more abstract notion than program, in a variety of ways, it is possible as well to have different programs in the same language for a particular algorithm

> ... differing in inessential respects (1984, p. 89).

There are a lot of difficulties here, some of which we have begun to address in the last chapter. While this section is meant mainly to present definitions of algorithms, let me point out two of the problems which will be addressed more completely later on. One is this use of abstraction and the implicit assumption that there is a level of algorithms above (or at least distinct from) the level of programs. At this point, Pylyshyn seems to be advocating a view of algorithms as the medium of the semantic level, perhaps, with programs the medium of the symbolic level. This would be quite a radical view of algorithms. Or is the algorithmic level supposed to be made up of these abstract algorithms? This is at odds with Newell's symbolic or program level on which Pylyshyn allegedly draws. The other immediate problem has to do with 'essential and inessential' differences in algorithms. This does affect Pylyshyn, because separating out the essence of an algorithm is one of his primary concerns.

In case there is any doubt that this is the correct interpretation of Pylyshyn, he states it quite explicitly (pp. 89-90):

> I suggest that the appropriate level of comparison [between computer models and psychological theories] corresponds roughly to the intuitive notion of the algorithm. An algorithm is related to a program approximately as a proposition is related to a sentence. The two are in a type-token relation. There are at least as many ways of expressing a proposition by some sentence -- or an algorithm by some program -- as there are different languages.

As propositions can in theory be expressed using formal languages, so too for algorithms. On the same page, then, Pylyshyn goes on to say,

> To specify what proposition or algorithm we have in mind, however, requires use of something like a canonical notation. Thus a proposition is identified with a sentence in some formal language (for example, predicate calculus) whereas an algorithm is identified with a program in some canonical programming language. (The emphasis is mine.)

After diverging from the standard definition of algorithm by removing the association with a particular language, Pylyshyn has now brought back not only a language but the language, at least for psychological theorising. His position is that

> an appropriate canonical language is crucial to the expression of mental processes at the required level of specificity. (p. 90)

This can be contrasted with the view advocated in this thesis, the view that the level of detail in which an algorithm is expressed, indeed what is essential about it and what is inessential, is relative to the question being asked and the context in which it is asked. The context must be allowed to be more precise than just cognitive psychology in general.

COMPARISON OF ALGORITHMS

One important reason for requiring an exact definition of algorithms is that such a definition can serve as a first step toward defining the equality of two algorithms or otherwise making comparisons between them. Of course, at the mathematical level, we already have a notion of equivalence (mathematical equivalence of functions and relations), but for any more detailed comparison than

that, something else is needed. Pylyshyn (1984) is after the same thing, and I have already drawn on his characterisation of this as 'strong equivalence'.

The first theoretical computer science approach that comes to mind is likely to be the theory of complexity of algorithms or theory of computation. This includes the 'big O' notation, as used in comparison of searching and sorting algorithms, paradigmatically. For example, a sequential search requires O(n), read 'on the order of n', time for searching through n unordered items. A sequential search is meant to be an algorithm on a standard von Neumann computer for finding a particular item in a list of like items, such as a personnel record for Jones in a list of personnel records. The time it takes to find a particular record on average is dependent on n, where n is the total number of records. More precisely, the average time to find a record would be the time it takes for n/2 comparisons (examining a record to find out if it is the record for Jones), plus some approximately constant overhead. Time varies proportionally to n, so typically O(n) would be used rather than O(n/2) in this case. Space as well as time can be measured in this manner. In the example of Jones and personnel records, the space requirements are also proportional to n, being n times the storage required for one record, plus some constant amount as before.

While these measures do not spell out algorithms in terms of a language for a canonical machine, they depend more or less explicitly on the relationship of standard programming languages to Turing machines. Here I am using Turing machine to mean the abstract ideal machine, as it will be defined shortly, and also the intuitive ideal machine defined in terms of tapes and operations rather than sets of quadruples. (This can be contrasted to a usage I have come across in the psychological

literature which is closer to my use of von Neumann machines, as in 'a VAX is a von Neumann machine', meaning a traditional sequential computer or one with minor variations.) In order to justify the assertion with which this paragraph opens, let us take a closer look at a formal definition of polynomial-time decidability from Davis and Weyuker (1983). This is a key concept in defining the intractability of NP-complete problems, such as the travelling salesman problem. Starting with the definition, then, as given on page 336, we can work backwards to establish the sort of comparison of algorithms that the theory of computation gives us. The definition is as follows:

> Definition. A language L on an alphabet A is said to be polynomial-time decidable if there is a Turing machine M which accepts L, and a polynomial p(n), such that the number of steps in an accepting computation by M with input $x \leq p(|x|)$. When the alphabet is understood, we write P for the class of polynomial-time decidable languages.

Davis and Weyuker give this formulation of a Turing machine (pp. 97-98):

> ... we imagine a device capable of various internal states. The device is, at any particular instant, scanning a square on a linear tape ... The combination of the current internal state with the symbol on the square currently scanned is then supposed to determine the next 'action' of the device. As suggested by Turing's analysis of the computation process ..., we can take the next action to be either 'printing' a symbol on the scanned square, or moving one square to the right or left. Finally, the device must be permitted to enter a new state.
>
> We use the symbols q1, q2, q3, ... to represent states and we write s0, s1, s2, ... to represent symbols which can appear on the tape, where as usual s0 = B is the 'blank'. By a quadruple we mean an expression of one of the following forms consisting of four symbols:

(1) qi sj sk ql,
(2) qi sj R ql,
(3) qi sj L ql

We intend a quadruple of type (1) to signify that in state qi scanning symbol sj, the device will print sk and go into state ql. Similarly, a quadruple of type (2) signifies that in state qi scanning sj the device will move one square to the right and then go into state ql. Finally a quadruple of type (3) is like one of type (2) except that the motion is to the left.

We now define a Turing machine to be a finite set of quadruples, no two of which begin with the same pair qi sj ...

The alphabet of a given Turing machine M consists of all the symbols si which occur in quadruples of M except s0.

We stipulate that a Turing machine always begins in state q1. Moreover, a Turing machine will halt if it is in state qi scanning sj and there is no quadruple of the machine which begins qi sj. With these understandings, and using the same conventions concerning input and output that were employed in connection with Post-Turing programs, it should be clear what it means to say that some given Turing machine M computes a partial function f on A* for a given alphabet A.

I have gone through these definitions in such excruciating detail in order to provide one exact definition of Turing machines and also to make clear just how detailed such a formulation must be for the formal theory of computation, in contrast to the looseness with which 'Turing machine' is frequently used. At the same time, it should be clear that the usual comparisons of algorithms or programs using terms such as 'O(n)' or 'polynomial time' are built upon the foundation of such an ideal mathematically defined machine. That these comparisons turn out to be useful in the practical implications for real programs is a testimony to the intimate relationship between current real computers and

the ideal Turing machine. In fact, as we have seen, the definition of a Turing machine is that of a completely formal mathematical object -- a set of quadruples; the intuitive machine, with its tape and scanner, provides a more accessible view of the mathematical object that somehow helps us to understand it.

In practice, as noted earlier, complexity times are often given in terms of operations on a different canonical machine: a register or von Neumann machine. Pylyshyn points out that table lookups with hash coding (a search method whose main virtue is constant time on a von Neumann machine) are O(n x n) on a Turing machine. The notion of strong equivalence of algorithms that he develops is based on complexity profiles, so according to the theory he outlines hashing cannot be implemented on a Turing machine!

Automata theory or theory of computation is related to formal semantics in that both provide a mathematical basis for reasoning about programs. Formal semantics approaches have been developed in order to clarify and solidify the foundations of programming languages; consequently they include a means of translating from a (usually high level) programming language into the 'semantic' domain of abstract mathematical entities or into a canonical ideal machine language. Following Pylyshyn's example, and drawing on Stoy (1977) and Gordon (1979), denotational and operational semantics will be considered in turn. Denotational semantics and to a certain extent operational semantics as well, are aimed at precise, machine independent concepts to support reasoning about programs. This reasoning is meant to be construed in a weak sense, as the concerns are with proving program correctness (equivalence of a program with a mathematical formulation of the function it is supposed to carry out) or program equivalence (whether or

not two different programs in the same or different languages compute the same mathematical function).

While Pylyshyn sees promise in denotational semantics for separating out the essential from the inessential, defining a stronger sort of equivalence of algorithms, this approach really is only concerned with weak mathematical equivalences. Stoy (pp. 12-13) gives the following description:

> We give 'semantic valuation functions', which map syntactic constraints in the program to the abstract values (numbers, truth values, functions, etc.) which they denote. These valuation functions are usually recursively defined: the value denoted by a construct is specified in terms of the values denoted by its syntactic subcomponents, and it is this emphasis on the values denoted by all these constructs that gives the approach its name. There may or may not be an obvious way of working out the results of these functions in any particular case: that is, the defining equations may or may not suggest a way of implementing the language.

All that is canonical is the mathematical translation function from a language to mathematical objects. Pylyshyn claims this says something about the 'appropriate level or grain of comparison' (1984, p. 91) of algorithms, but it is a side effect of the method that this level or grain is very close to the syntax of the (usually high level) language being investigated, if indeed the equations are taken as more than abstract objects.

While recognising that equating denotational semantics with high level language design and analysis on the one hand, and equating operational semantics with implementation concerns, on the other, is overly simplistic, Stoy himself succumbs to the confusion. He states (p. 24) that

> Operational definitions ... contain extra implementation details which, unless they are explicitly relevant to a particular problem, serve merely to complicate further an area which is already complex.

But this is not necessarily true because it is possible (as indicated in the previous chapter) to define algorithms at any level of detail. Any such algorithm can be seen as running on some ideal machine. This concept of extending the range of what is usually meant by ideal machine will be elaborated in the later sections of this chapter and formalised in Chapter V.

Perhaps operational semantics are more appropriate for our purpose of explicating algorithms and algorithmic equivalence. In the world of theoretical computer science, one gets the idea that operational semantics is the old-fashioned way of looking at programs. Using such a method, a language is given its formal semantics by providing a translation to a (typically) lower level ideal machine language, likely to be the standard machine code for a particular machine, or perhaps something more portable. (These days it would probably be 'C'). PL/1 and ALGOL 68 were defined in this way. Stoy (p. 19) remarks on the operational approach

> that the method tends to be regarded as one which gives the result only of specific computations. Starting with some particular program, and some particular input data one may 'crank the handle' of the defined abstract machine and obtain the particular final result. This job is, however, precisely what the computer is for. It is much more useful for us to be able to consider the class of all the computations that can possibly be evoked by the program, or even by a class of programs. When we do this, we are in effect considering the function that the program and its implementation together define. We are, in fact, moving towards a function approach.

For the purposes of cognitive science, however, insofar as the purposes include describing and explaining at various levels the activities and states of a complex system, it is inadvisable to stray too far from the actual system in question. Of course, as we have seen, the abstract mathematical approach can tell us something, but only a limited amount. Operational semantics, therefore, seems closer to what we need, at least as the basis for equivalence of states and perhaps algorithms. (At the same time there is nothing preventing expressing generalisations in mathematical functional terms where appropriate.) Still, even operational semantics is too weak! While Pylyshyn sees in it a means of making stronger comparisons than strictly mathematical ones, the comparisons are only weak mathematical ones, unless the states of the canonical machine are abstractions (in a precise sense that will be elaborated in the next sections) of the system that is being described. While operational semantics distinguishes between, for example, an algorithm that computes 2x by adding x to itself and one that computes 2x by calculating x + x + 7 - 7, comparisons can only be made at the one level defined by computations for the one canonical machine. These weak comparisons are perfectly suited to the computational theoretical goals of correctness and weak equivalence proofs, and this is confirmed by the sensible shift to the less operational versions of programming language semantics.

The feature of having a single ideal machine or canonical form in terms of which to compare algorithms is common to all the approaches surveyed here. This is a severe drawback for the purposes of comparing algorithmic descriptions of complex systems, because it forces all comparisons to be made at a single level of detail. Except in circumstances where both algorithms can be

seen at an appropriate level as running on the one canonical machine, one or both of the algorithms/descriptions must be distorted to fit the canonical form. Of course, as has been noted again and again, this is not a problem for the original purposes for which these techniques were developed, i.e. complexity and tractability of algorithms on von Neumann machines or proofs of correctness or mathematical equivalence of programs and functions.

A NEW DEFINITION OF ALGORITHMS

First, a comment about terminology is in order. I have chosen to continue to use the term 'algorithm', because I think the new definition about to be presented captures the spirit with which the term is currently used throughout much of cognitive psychology and even computer science. From now on this usage shall be distinguished from the definitions given earlier of algorithms as effective procedures or programs in some particular language. These shall hereafter be called by their more narrowly defined names: effective procedures or programs, whichever is appropriate.

For an example, let us return to the exclusive-or function. In particular, to be clear, consider the following effective procedure in PASCAL:

```
PROGRAM XOR;
VAR X, Y, Z:  INTEGER;
BEGIN
     READLN(X);
     READLN(Y);
     IF (X=Y) THEN
          Z:=0
     ELSE
          Z:=1;
     WRITELN(Z)
END.
```

The new definitions and concepts will be presented initially in terms of this example and, where appropriate, contrasted to concepts from computational theory. One of the most basic concepts is the state of an ideal machine. To convey the idea, the example program might describe a machine that starts off in the state shown by

```
X: U
Y: U
Z: U
NEXT INSTRUCTION:  READLN(X);
```

In front of the colons are labels; values follow the colons. The value 'U' stands for undefined. I am using it here as a shorthand to represent what may either be a specific value which represents an undefined value (such as -11111111) or an arbitrary value. Davis and Weyuker (p. 23) define a slightly different state of a program P, relying on their rigorously defined languages, as

> a list of equations of the form V = m, where V is a variable and m is a number, including exactly one equation for each variable that occurs in P.

This definition differs from the one I want to give, and it is instructive to emphasise the differences. Primarily, Davis and Weyuker's state includes exactly those variables and values mentioned in the program, no more and no less. A state is therefore tied to a particular program. This is clear in their terminology, for they say 'state of a program P' or 'state of P'. On the other hand, one of my main objectives is to define a state independent of a particular program. It is best thought of as the state of an ideal machine, such as a Turing machine, but allowing other possibilities as well. Which values are included and what labels they are given then depend on the ideal machine and the context; this should become clear as we progress.

The first definition I would like to suggest is that of an ideal machine state or simply a state as a set of ordered pairs, where each ordered pair consists of a label followed by a value. Informally, such a state will be shown, as above, as a collection of labels and values separated by colons. The set notation for the example is {(x,u), (y,u), (z,u), (next instruction, READLN(X))}. A descriptive term for a state is a snapshot, but once again care should be taken to distinguish my use of 'snapshot' as a synonym for an ideal machine state from the theoretical computer science usage. Davis and Weyuker (p. 24)

> define a snapshot or instantaneous description of a program P of length n to be a pair (i,sigma) where $1 \leq i \leq n + 1$, and sigma is a state of P. (Intuitively the number i indicates that it is the ith instruction which is about to be executed; $i = n + 1$ corresponds to a 'stop' instruction.)

It should be noted that the state illustrated above is not a state of a program at all in Davis and Weyuker's terms, though it is (a notational variant of) a snapshot.

Of course the important distinctions are not those of terminology. The first crucial move is the dissociation of states from a particular language or a particular program. The second is the (almost) complete flexibility over what counts as a label and a value and which labels and values are to be included in a particular state; as we shall see later, this will facilitate viewing algorithms from many different levels of detail. Notice also that the 'NEXT INSTRUCTION' label-value pair has no special status in the new definition. The idea is to generalise the use of the ideal Turing machine as a basis for theoretical computer science to a multiplicity of different but related ideal machines as a basis for theoretical cognitive science.

The second definition is for what I shall call an algorithmic sequence; that is a sequence of states defined over the same set of labels through time. For example the program given earlier gives rise to the following algorithmic sequence, as one possibility. (Each group of label-value pairs is a state; time runs from top to bottom.)

X: U
Y: U
Z: U
NEXT INSTRUCTION: READLN(X)

X: 0
Y: U
Z: U
NEXT INSTRUCTION: READLN(Y)

X: 0
Y: 1
Z: U
NEXT INSTRUCTION: X = Y?

X: 0
Y: 1
Z: U
NEXT INSTRUCTION: Z = 1

X: 0
Y: 1
Z: 1
NEXT INSTRUCTION: WRITELN(Z)

X: 0
Y: 1
Z: 1
NEXT INSTRUCTION: U

The constraint on states to be 'defined over the same set of labels' ensures that labels are consistent across states. In formalising these definitions in Chapter V, we will speak of states and algorithms over sets of labels. The ideal machine is better thought of as comprising not only labels, but also possible state sequences or, alternatively, transitions between states. In short an ideal machine is just another name for an algorithm. The definition of an algorithmic sequence can be contrasted to Davis and Weyuker's computation (p. 25):

> A computation of a program P is defined to be a sequence (i.e. a list) s1,s2, ..., sk of snapshots of P such that si + 1 is the successor of si for i = 1,2, ..., k-1 and sk is terminal.

Here again the main difference is what is implicit in the latter definition. Each snapshot must contain all variables mentioned in P, along with the next instruction. The term 'successor' is also defined in terms of the next instruction, with explicit cases identifying the possible changes from one state to the next in terms of the next instruction from the last snapshot. The algorithmic sequence shown above is actually very close to a computation in this sense, with respect to the exclusive-or program given earlier. To avoid confusion, here are two examples of sequences which may arise from the execution of the same program but would not be considered computations by Davis and Weyuker. (Separate sequences are separated by a horizontal line.)

X: U
Y: U
Z: U

X: 0
Y: 1
Z: U

X: 0
Y: 1
Z: 1

X: 0
Y: 1
Z: U

X: 0
Y: 1
Z: 1

Just as two computations are presumably considered the same if they are identical, two algorithmic sequences are the same if and only if they are equal by usual n-tuple equality, giving us our third definition. Once again, there is no language automatically associated with such a sequence. Given a certain level of detail and perspective along with the input, a program may be seen as giving rise to an algorithmic sequence, or as abbreviating a set of them, given a set of inputs. There is another informal sense in which two sequences might be considered to be the same. That is when one is an abstraction of the other or when they have a common abstraction or implementation.

More generally, an algorithm is a set of algorithmic sequences defined over the same set of labels. If a mathematical definition of a function is given as a set of ordered pairs, then an algorithm so defined is a set of sequences of states which includes each input in the first state of some sequence and the output is part of the final state of that sequence. Although we are defining them here as sets and sequences and other mathematical entities, the equivalence of interest at this level is stronger than that of the mathematical functional level which only concerns input and output. There is also some notion of time (in number of steps) and space (in number of values), although it is not absolute. On this view, both the algorithmic and mathematical levels of the second chapter deal with mathematical objects. The difference lies in the sort of equivalence at each level: weak functional, perhaps behavioural, equivalence at the top level and stronger, albeit still mathematical, 'algorithmic' equivalence at the middle level.

We now have the building blocks for a rigorous definition of abstraction of algorithms or levels of algorithmic

abstraction in terms of detail. While talk of levels and abstraction permeates cognitive science, to my knowledge there has been little attempt at a formal definition of this kind of abstraction. Another sort of formal abstraction is the case where algorithms are equated with programming languages and there is a hierarchy of languages, such as a graphics language built on top of FORTRAN built on top of VAX assembler language. In a case like this, descriptions are constrained to the available languages, and even then the relationship between the states described at a higher level and those at the lower level may be tenuous, depending on the compilation or translation process. For the purposes of accurate description at various levels of a given complex physical system and the states through which it passes, something different is required.

Illustrations of abstraction for algorithmic sequences have already been presented. Recall the six-step illustration above, which showed a sequence of states corresponding to the PASCAL program for exclusive-or. The next example sequence shown was an abstraction of this, having fewer states and fewer label-value pairs, and the next gave a further abstraction, presenting the algorithmic sequence with little more detail than the input and output values. But what exactly are the legal abstraction operations? For now, let us look at another example. The complete definitions of abstraction will be given in Chapter V. Here is a sequence consisting of six steps defined over the set of labels {0, 1, 2, 3, 4}.

```
0: U        2: U
1: U        3: U
            4: U

0: 0        2: U
1: U        3: U
            4: U

0: 0        2: U
1: 1        3: U
            4: U

0: 0        2: 0.96
1: 1        3: 0.02
            4: U

0: 0        2: 0.98
1: 1        3: 0.02
            4: 0.98

0: 0        2: 0.98
1: 1        3: 0.01
            4: 0.99
```

The labels are simply numbers. (It is not crucial to the point being made here, but the algorithmic sequence might be a description of a typical connectionist model of exclusive-or. This underlying model will be considered more fully later on.) The results of one possible legal abstraction operation are shown in the following. ('0' has been changed to 'X', '1' to 'Y', '2' to 'X OR Y', '3' to 'X AND Y' and '4' to 'OUTPUT'.)

```
0: U        2: U
1: U        3: U
            4: U

0: 0        2: U
1: U        3: U
            4: U

0: 0        2: U
1: 1        3: U
            4: U

0: 0        2: 1
1: 1        3: 0
            4: U

0: 0        2: 1
1: 1        3: 0
            4: 1

0: 0        2: 1
1: 1        3: 0
            4: 1
```

This operation is rounding; the values associated with labels 2, 3 and 4 have been rounded off. Another valid abstraction operation is the changing of labels, as demonstrated in the following:

```
X: U        X OR Y: U
Y: U        X AND Y: U
            OUTPUT: U

X: 0        X OR Y: U
Y: U        X AND Y: U
            OUTPUT: U
```

```
X: 0        X OR Y: U
Y: 1        X AND Y: U
            OUTPUT: U

X: 0        X OR Y: 1
Y: 1        X AND Y: 0
            OUTPUT: U

X: 0        X OR Y: 1
Y: 1        X AND Y: 0
            OUTPUT: 1

X: 0        X OR Y: 1
Y: 1        X AND Y: 0
            OUTPUT: 1
```

I think of this as a sort of 'lateral abstraction', because it seems more like a shift on the same level than a move to a higher level of abstraction. There is no change in the amount of detail, only in the perspective. The change is quite vivid, showing not only that this specific algorithm may be seen as a calculation of the exclusive-or function for inputs zero and one, but also saying something about how the output is reached. In particular, it shows the calculation of 'X OR Y' and 'X AND Y' as intermediate steps. Getting only slightly ahead of ourselves, one can begin to see how a state-based algorithmic sequence such as this might be used to compare such diverse models as traditional and connectionist ones. At this level of abstraction, for example, a distinction can be made between the states generated by effective procedures for calculating exclusive-or by first calculating 'or' and 'and' and other versions, such as a procedure which uses an equality check ('X XOR Y' is true if and only if 'X = Y' is false; this will work as long as the inputs are restricted to ones and zeroes).

In the next sequence the labels have been changed again ('X' to 'INPUT1' and 'Y' to 'INPUT2') and now only three label-value pairs are shown in each state, illustrating the abstraction operation of 'selection of values'.

INPUT1: U
INPUT2: U
OUTPUT: U

INPUT1: 0
INPUT2: U
OUTPUT: U

INPUT1: 0
INPUT2: 1
OUTPUT: U

INPUT1: 0
INPUT2: 1
OUTPUT: U

INPUT1: 0
INPUT2: 1
OUTPUT: 1

INPUT1: 0
INPUT2: 1
OUTPUT: 1

Next we have the results of applying the 'selection of steps' operator, followed by the result of an application of the 'grouping of values' operation.

INPUT1: 0
INPUT2: 1
OUTPUT: U

INPUT1: 0
INPUT2: 1
OUTPUT: 1

INPUT : 0,1
OUTPUT: U

INPUT : 0,1
OUTPUT: 1

To review, the abstraction operations that have been introduced are changing labels, rounding, selection of states, selection of values and grouping of values. Another kind of abstraction for algorithms, as opposed to sequences, is 'selection of sequences'. One algorithm is an abstraction of another if the first is a subset of the second. All of these combine to give a coarser view at the top, most abstract level and a finer, more detailed view at the bottom level. In addition, taken together they give an explicit path of derivation relating the various levels. Although this introduction has been in terms of abstraction, the process is intended to be reversible, giving levels of abstraction viewed one way and details or explanations (or implementations) when viewed the other way. The added details of a lower level can be seen as an explanation if they provide the answer to a question in terms of the higher level, such as 'How is exclusive-or calculated?' or 'Are the inputs always received in a certain order?'

Before developing some more complete examples, it should be noted that some possible abstraction operations are not allowed. In general, these are movements to a less

coarse description, or to a description that contradicts a lower level description. The idea is that details are removed, not added. So while the removal of steps or states is a legal abstraction operation, the addition of states is not a valid move. Changing labels is neutral in this respect. Providing more precision for a value (the opposite of rounding) is not allowed in general. The abstraction/implementation relationship is defined more completely and precisely in Chapter V.

DETAILED EXAMPLES OF EXCLUSIVE-OR ALGORITHMS AND ABSTRACTION

To see how this new perspective on algorithms facilitates their comparison, consider four simple mathematically equivalent ways of computing exclusive-or. Before converting them to fit the new framework, we will look at some more traditional presentations in the form of PASCAL programs, Turing machines and connectionist diagrams.

First, there is the PASCAL version we have already seen; it shall be referred to as 'the first PASCAL version' or 'the equality check version'. For convenience it is repeated here.

```
PROGRAM XOR;
VAR X, Y, Z: INTEGER;
BEGIN
     READLN(X);
     READLN(Y);
     IF (X=Y) THEN
          Z:=0
     ELSE
          Z:=1;
     WRITELN(Z)
END.
```

This has already been contrasted to a possible PASCAL version (henceforth 'the second PASCAL version', or 'the Boolean version') which is now made explicit.

```
PROGRAM XOR;
VAR X, Y, Z: INTEGER;
VAR X2, Y2, Z2:  BOOLEAN;
BEGIN
     READLN(X);
     READLN(Y);
     IF (X=0) THEN
          X2:=FALSE
     ELSE
          X2:=TRUE;
     IF (Y=0)THEN
          Y2:-FALSE
     ELSE
          Y2:=TRUE;
     Z2:=(X2¦Y2) & NOT (X2 & Y2);
     IF (Z2=FALSE) THEN
          Z:=0
     ELSE
          Z:=1;
     WRITELN(Z)
END.
```

The third example is somewhat different. Like the PASCAL programs, it is arguably an effective procedure. In fact, it is a Turing machine in Davis and Weyuker's terminology, or it can be seen as a Turing machine program.

For the exclusive-or example, the tape begins at state q1 with the input in two squares and the 'tape head' positioned at the first input. At the completion of the computation, the device is in state q9 and the tape head

is positioned at the square containing the output, with the rest of the tape blank. Here, along with some comments to help clarify what is going on, is the entire Turing machine or Turing machine program.

```
q1 0 R q2  In case the first item on the tape is a
           zero (0), move to the right and into the
           state of 'we have a 0 and a ...?'
q1 1 R q3  In case the first input on the tape is a
           one (1), move to the right and into the
           state of 'we have a 1 and a ...?'
q2 0 B q5  Here we know we have a 0 first and we
           learn there is a 0 in the next position as
           well.  The proper final answer is 0, so
           all that needs to be done is to blank out
           the second 0, which is accomplished in
           this instruction, and to reposition the
           tape head;  this will happen as a result
           of the next instruction starting with q5 B
           ...
q2 1 B q4  Again, we know we have a 0 first, but in
           this case we learn there is a 1 in the
           next position.  The proper answer is 1,
           so we must alter the first character.
           But first the second position must be
           blanked out.
q3 0 B q5  Here we had a 1 and now meet a 0.  The
           second input is blanked out and we go to
           the state where the tape head is about to
           be repositioned at the first input, which
           is identical to the output.
q3 1 B q6  Here we had a 1 and now encounter another
           1.  The answer will be 0, so we must
           alter the first character after blanking
           out the second.
```

```
q5 B L q9 This  is the case where all that remains
          is  to  reposition  the  tape  head  at
          the beginning;  the initial square already
          contains the correct answer.
q4 B L q7 Here we go back to the original square,
          but it remains to change it from a 0 to a
          1.
q6 B L q8 Similarly, we return to the first input,
          but now it must be changed from a 1 to a
          0.
q7 0 1 q9 This instruction changes the first square
          from a 0 to a 1.
q8 1 0 q9 This instruction changes the first square
          from a 1 to a 0.
```

A state transition diagram may help to elucidate this sort of program. In Figure 3.1 the states are represented by circles containing the qi's. The possible characters being scanned and the action to be taken are given along the arcs, which represent the possible transitions between states. The Turing machine is considered to halt when it reaches a state and tape square for which there is no further transition, in this case q9. It may also help to note these descriptions of each state.

In q1, we are in the initial state.
In q2, we have read a 0 so far.
In q3, we have read a 1 so far.
In q4, we have read a 0 and a 1, so the answer will be 1.
In q5, we have either two 0's or a 1 and a 0. In either case the answer is already correct in the first square.
In q6, we have two 1's, so the answer will be 0.

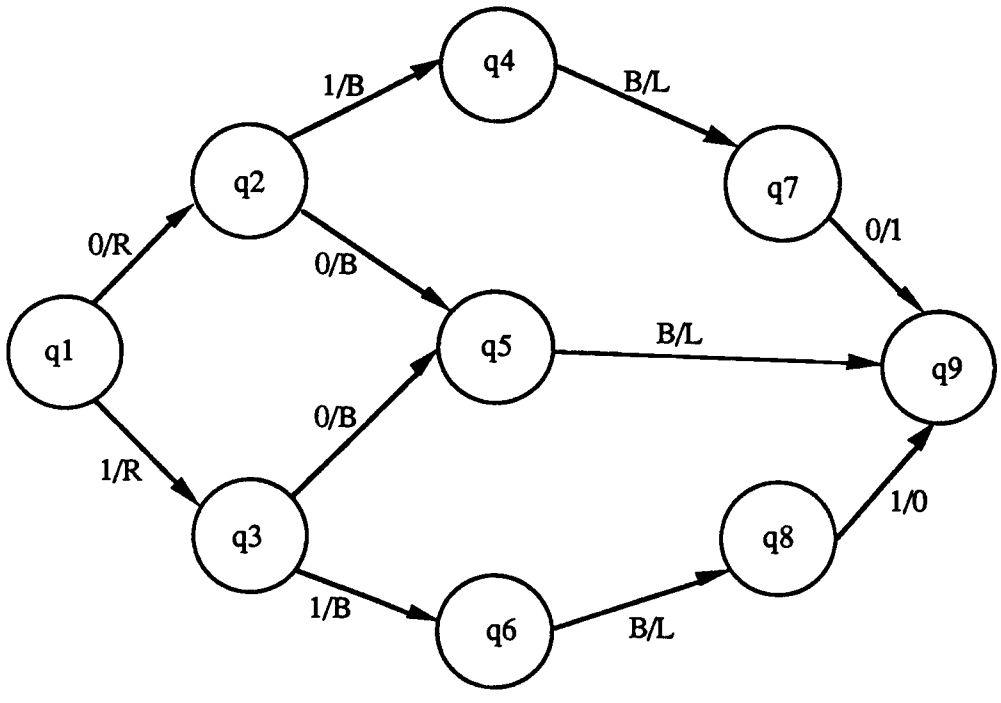

FIGURE 3.1

In q7, we have a 0 and a 1, the second of which has been blanked out. All that remains is to change the 0 to a 1.
In q8, similarly, we have two 1's, the second of which has been erased. All that is left is to change the 1 to a 0.
In q9, we are in the final state.

The fourth and fifth exclusive-or examples share a typical connectionist diagram, as shown in Figure 3.2. The large and growing literature on connectionism (otherwise known as parallel distributed processing or neural nets, among other things) is introduced by Rumelhart and McClelland (Rumelhart, McClelland, and the PDP Research Group, 1986 -- Chapter 2 of this volume, 'A General Framework for Parallel Distributed Processing' is particularly appropriate in the context of these examples; also see McClelland, Rumelhart, and the PDP Research Group, 1986; McClelland and Rumelhart, 1988).

The circles or 'nodes' or 'units' represent simple processors. In this example, each one will compute the function which sums up weighted inputs, and then outputs a 1 or a 0, depending on whether the weighted sum exceeds a threshold or not. Typically, all the processors in a connectionist model compute the same function. Also typically, it is a nonlinear threshold or smoothed threshold-like function.

The input and output for each processor are represented by arrows. Where these arrows connect two nodes, they are sometimes called connections. Since it is usually the configuration of these connections and their weights (more on these in a moment) that are emphasised in the current research on these systems, it is not unreasonable to label these models as 'connectionist'.

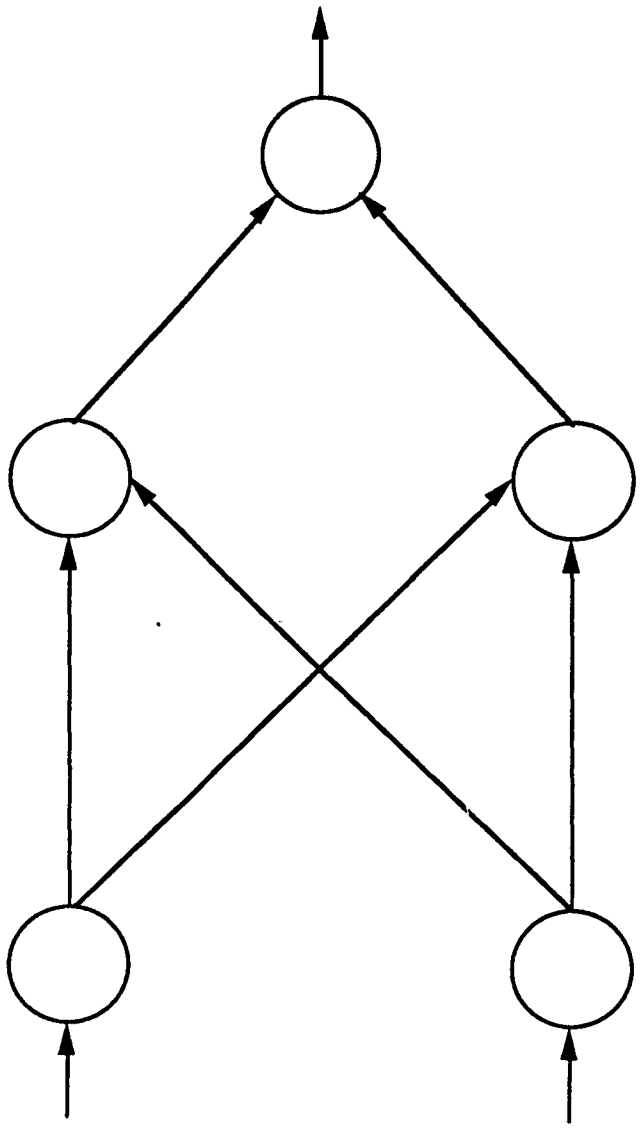

FIGURE 3.2

Nodes or units which receive input from outside the system (the bottom two in Figure 3.2) are called input nodes or input units. Similarly, those that send output outside the system are called output nodes or output units. Together they are called 'visible' nodes or units. In contrast are 'hidden' units, such as the middle two in the figure. These hidden units are worth noting, because the recent resurgence of interest in connectionist systems is largely due to the discovery of 'learning algorithms' for systems with hidden units. Such systems are much more powerful than those without hidden units, in terms of the functions which they are capable of computing.

Figures 3.3 and 3.4 show the same diagram, but this time including the weights and thresholds which distinguish two different algorithms at certain levels of description. (Figure 3.4 is taken from Rumelhart, Hinton and McClelland, 1986, p. 64 in Rumelhart, McClelland and the PDP Research Group, 1986). The numbers alongside the connections in these figures represent the weights. Each processor in these examples (again this is typical though by no means required for a connectionist system), as stated earlier, takes weighted inputs. That is, before applying the threshold function, the input values are multiplied by their weights. This may be part of the transmission of the value or part of the function itself. It is generally held that 'programming' a connectionist system amounts to adjusting these weights, while the connection configuration and activation and output functions remain fixed for a particular system. (Even for such a simple function as exclusive-or, it is no small task to determine the correct settings for the weights.) In fact it is so difficult in general that it is usually done automatically, using learning algorithms, as mentioned

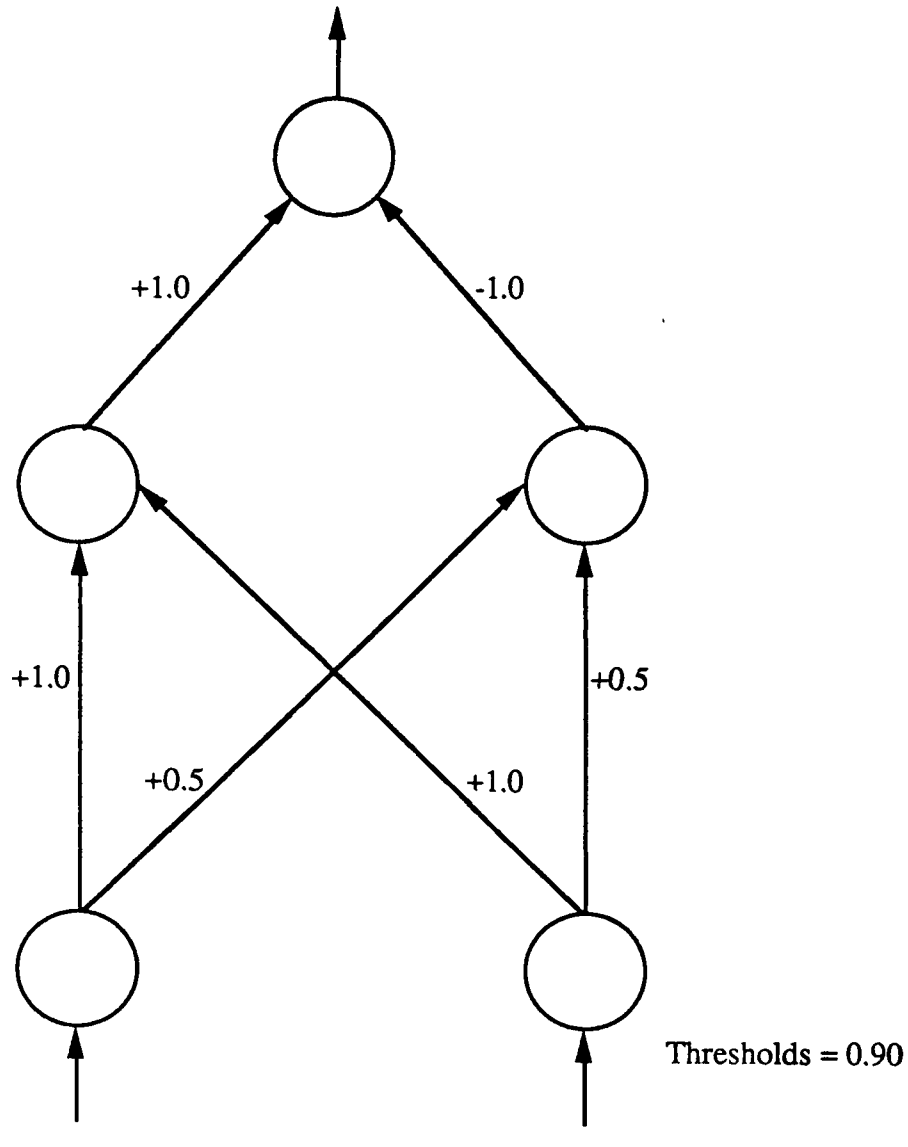

FIGURE 3.3

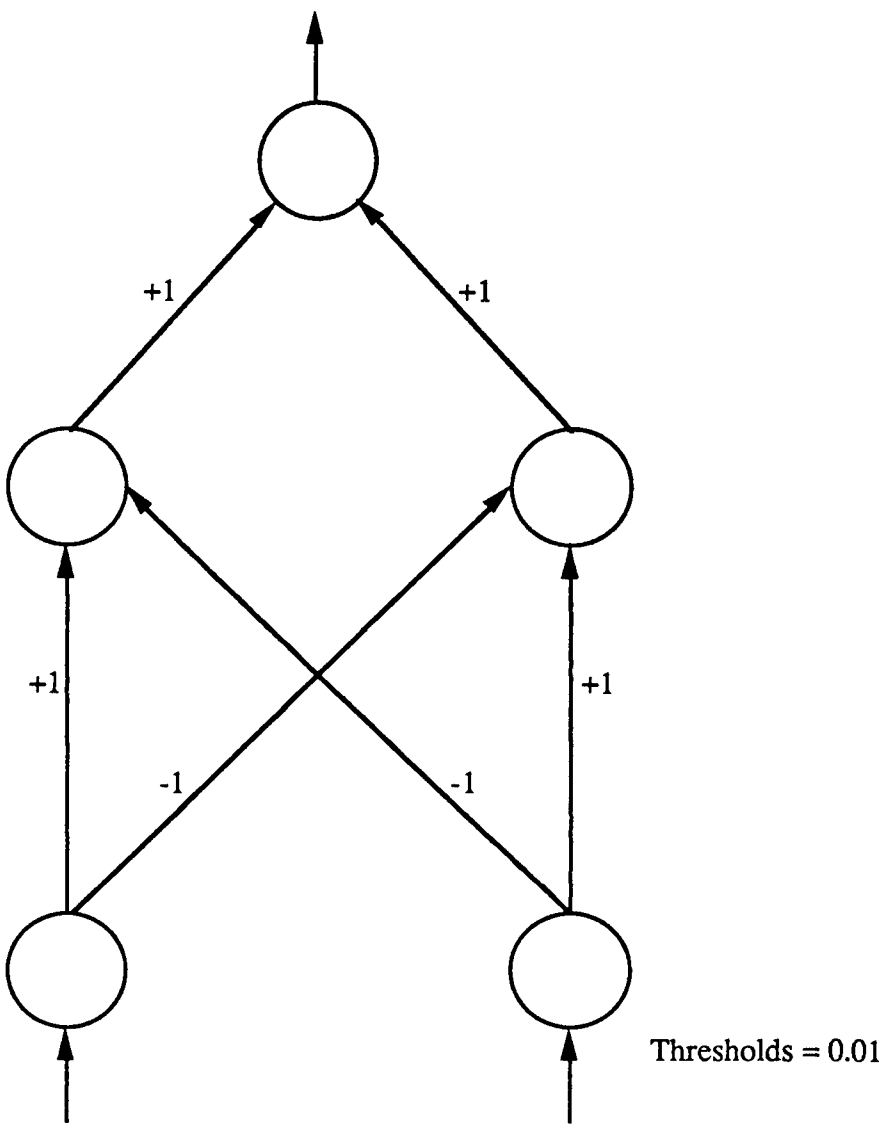

FIGURE 3.4

above. In this way, the automatic program or learning algorithm is given the configured system, along with the desired inputs and outputs, in the example the mapping (0,0) --> 0, (0,1) --> 1, etc. It then calculates the weights required to produce this output. A large proportion of connectionist research is concerned with these learning procedures, their analysis and extensions, and the results of applying the resultant 'trained' systems to hitherto untried inputs. There is also a great deal of interest in relating these learning methods to human psychological and biological learning capacities. The emphasis on learning, however, will not be shared by this thesis. There is enough to be said about the description of these complex systems in their more stable, 'programmed' states. Theories of cognitive psychology and neuroscience may well benefit from research into learning algorithms, but it is useful to consider less complicated cases as well.

The systems in Figures 3.3 and 3.4, then, take inputs that can be represented as zeroes and ones and produce outputs represented as zeroes and ones, computing exclusive-or. While calling the Turing machine example a 'program' may seem to stretch the term a little, it is even more difficult to justify the use of 'program' or 'effective procedure' to describe these systems. Such systems are built or simulated using conventional hardware with, e.g., a subroutine for each node, yet they are frequently contrasted to von Neumann or Turing-machine based computation. Nevertheless, it is not completely unnatural to refer to these as rather different 'algorithms' for computing exclusive-or. Of course, it is one of the fundamental motivations of this thesis to develop a framework for comparing theories and algorithms of all these different types. Using the new definitions of algorithms and algorithmic abstraction with these examples, let us see how far we get.

COMPARISON OF EXAMPLES

Because the explanation and description of complex physical systems are our main concerns, let us assume the various descriptions given as examples of exclusive-or are accurate descriptions of physical systems in that they are abbreviations for states actually attainable by those systems, described at some level of detail according to the precise definitions given in this thesis. Even so, what is the exact algorithm, in terms of state sequences, is open to debate without the actual system -- or a lower level description -- to check. Consider the algorithmic sequence for inputs 0 and 1 using the first PASCAL version. As in the earlier discussion of this example, one level of description would give this sequence as follows.

```
X: U
Y: U
Z: U
NEXT INSTRUCTION:  READLN(X)

X: 0
Y: U
Z: U
NEXT INSTRUCTION:  READLN(Y)

X: 0
Y: 1
Z: U
NEXT INSTRUCTION:  X = Y?

X: 0
Y: 1
Z: U
NEXT INSTRUCTION: Z := 1
```

```
X: 0
Y: 1
Z: 1
NEXT INSTRUCTION: WRITELN(Z)

X: 0
Y: 1
Z: 1
NEXT INSTRUCTION: U
```

Under this view of the algorithm, it is clear that the next instruction is explicitly represented somehow (notice we do not say how) in the system. It may be all in one place, as in a next-instruction register, or it may be pointed to by such a register with the entire program explicitly stored. All that is required of values is that they be directly measurable from the physical system, or built up through the legal abstraction operations. A value may even be a measurement of a 'process' represented by some pattern of activity or it may be inactive, represented by an instruction on a sheet of paper, perhaps the instruction for a person or a device with an optical scanner. We need this stronger description of states in order to talk sensibly about the system as being in a state such as the state of comparing the two values. Viewing the program as software, there is no intrinsic constraint that forces it to be an accurate description of the states of a physical system -- even if that system is a computer executing a program generated from that software. The compilation of the software into the basic operations of a machine could conceivably produce a system that goes through states such as the following. It reads a part of the input character for X, then it reads a part of the input character for Y. One is curved and one is straight, so it outputs '1'. In that case, I claim, it

is wrong to say that the system gets one complete input, then the other, in sequence. In fact it may not read the entire input at all. Software, under this view, is merely a more or less accurate theory of the workings of a complex system, a computer. It is at least a correct predictive theory (predicting the correct output given the input), but it may not be a very good explanatory theory (that is, a theory which is stronger than mathematical equivalence, one that explains something about how the output is reached). The accuracy of software in a traditional computer system as a theory of the states through which the system passes depends on the compilation (or interpretation in the compiler theoretic sense) and execution of the result.

Comparison of Algorithms Represented by the Two PASCAL Programs

At the level of input and output, the two PASCAL versions of exclusive-or are identical, since both give rise to the sequence.

X: U
Y: U
Z: U

X: 0
Y: 1
Z: U

X: 0
Y: 1
Z: 1

Further similarity can be seen, since they get their inputs in the same order and they retain the input through to the end, as below

X: U
Y: U
Z: U

X: 0
Y: U
Z: U

X: 0
Y: 1
Z: U

X: 0
Y: 1
Z: 1

If we go into much more detail, however, the differences begin to emerge, even without considering stored instructions. For example, under the usual implementations of PASCAL, at some point the logical value 'X = Y' will be computed for the first version, as will the logical values of 'X2 OR Y2', 'X2 AND Y2' and so on for the Boolean version. A comparison at such a level follows, making use of the PASCAL variables and logical equations to determine labels and values in the ideal machine states.

Version 1:

X: U	X = Y: U
Y: U	Z: U
X: 0	X = Y: U
Y: U	Z: U
X: 0	X = Y: U
Y: 1	Z: U

```
X: O          X = Y: FALSE
Y: 1          Z: U

X: O          X = Y: FALSE
Y: 1          Z: 1
```

Version 2:

```
X: U      X = 0: U          XZ: U
Y: U      Y = 0: U          YZ: U
X2 OR Y2: U
X2 AND Y2: U
NOT (X2 AND Y2): U
(X2 OR Y2) AND NOT (X2 AND Y2): U
Z2: U     Z2 = FALSE: U     Z: U


X: 0      X = 0: U          X2: U
Y: U      Y = 0: U          Y2: U
X2 OR Y2: U
X2 AND Y2: U
NOT (X2 AND Y2): U
(X2 OR Y2) AND NOT (X2 AND Y2): U
Z2: U     Z2 = FALSE: U     Z: U


X: 0      X = 0: U          X2: U
Y: 1      Y = 0: U          Y2: U
X2 OR Y2: U
X2 AND Y2: U
NOT (X2 AND Y2): U
(X2 OR Y2) AND NOT (X2 AND Y2): U
Z2: U     Z2 = FALSE: U     Z: U
```

```
X: 0        X = 0: TRUE          Z2: U
Y: 1        Y = 0: U             Y2: U
X2 OR Y2: U
X2 AND Y2: U
NOT (X2 AND Y2): U
(X2 OR Y2) AND NOT (X2 AND Y2): U
Z2: U       Z2 = FALSE: U        Z: U
```

```
X: 0        X = 0: TRUE          X2: FALSE
Y: 1        Y = 0: U             Y2: U
X2 OR Y2: U
X2 AND Y2: U
NOT (X2 AND Y2): U
(X2 OR Y2) AND NOT (X2 AND Y2): U
Z2: U       Z2 = FALSE: U        Z: U
```

```
X: 0        X - 0: TRUE          X2: FALSE
Y: 1        Y = 0: FALSE         Y2: U
X2 OR Y2: U
X2 AND Y2: U
NOT (X2 AND Y2): U
(X2 OR Y2) AND NOT (X2 AND Y2): U
Z2: U       Z2 = FALSE: U        Z: U
```

```
X: 0        X = 0: TRUE          X2: FALSE
Y: 1        Y = 0: FALSE         Y2: TRUE
X2 OR Y2: U
X2 AND Y2: U
NOT (X2 AND Y2): U
(X2 OR Y2) AND NOT (X2 AND Y2): U
Z2: U       Z2 = FALSE: U        Z: U
```

```
X: 0        X = 0: TRUE          X2: FALSE
Y: 1        Y = 0: FALSE         Y2: TRUE
X2 OR Y2: TRUE
X2 AND Y2: U
NOT (X2 AND Y2): U
(X2 OR Y2) AND NOT (X2 AND Y2): U
Z2: U       Z2 = FALSE: U        Z: U
```

```
X: 0        X = 0: TRUE          X2: FALSE
Y: 1        Y = 0: FALSE         Y2: TRUE
X2 OR Y2: TRUE
X2 AND Y2: FALSE
NOT (X2 AND Y2): U
(X2 OR Y2) AND NOT (X2 AND Y2): U
Z2: U       Z2 = FALSE: U        Z: U
```

```
X: 0        X = 0: TRUE          X2: FALSE
Y: 1        Y = 0: FALSE         Y2: TRUE
X2 OR Y2: TRUE
X2 AND Y2: FALSE
NOT (X2 AND Y2): TRUE
(X2 OR Y2): U
(X2 OR Y2) AND NOT (X2 AND Y2): U
Z2: U       Z2 = FALSE: U        Z: U
```

```
X: 0        X = 0: TRUE          X2: FALSE
Y: 1        Y = 0: FALSE         Y2: TRUE
X2 OR Y2: TRUE
X2 AND Y2: FALSE
NOT (X2 AND Y2): TRUE
(X2 OR Y2) AND NOT (X2 AND Y2): TRUE
Z2: U       Z2: FALSE: U         Z: U
```

```
X: 0       X = 0: TRUE          X2: FALSE
Y: 1       Y = 0: FALSE         Y2: TRUE
X2 OR Y2: TRUE
X2 AND Y2: FALSE
NOT (X2 AND Y2): TRUE
(X2 OR Y2) AND NOT (X2 AND Y2): TRUE
Z2: TRUE  Z2 = FALSE: U        Z: U
```

```
X: 0       X = 0: TRUE          X2: FALSE
Y: 1       Y = 0: FALSE         Y2: TRUE
X2 AOR Y2: TRUE
X2 AND Y2: FALSE
NOT (X2 AND Y2): TRUE
(X2 OR Y2) AND NOT (X2 AND Y2): TRUE
Z2: TRUE  Z2 = FALSE: FALSE    Z: U
```

```
X: 0       X = 0: TRUE          X2: FALSE
Y: 1       Y = 0: FALSE         Y2: TRUE
X2 OR Y2: TRUE
X2 AND Y2: FALSE
NOT (X2 AND Y2): TRUE
(X2 OR Y2) AND NOT (X2 AND Y2): TRUE
Z2: TRUE  Z2 = FALSE: FALSE    Z: 1
```

Of course, considering effective procedures as algorithms includes assumptions about the order of execution (here, generally, top to bottom, right to left, with the usual precedence of parentheses and logical operations). This is one of many ways in which the definition of algorithms as sequences of states overcomes the ambiguity of a list of instructions. Rather than saying this program is an algorithm at the theoretical level and can be implemented in many ways, the order of operations (e.g.) being inessential to the algorithm, the view presented here is that the various sequences of states derivable from this

program may or may not be important for a given explanation, depending on the context and question. They may or may not be accurate with respect to a given physical system.

So far we have only considered the algorithmic sequence for '0 xor 1'. Looking at the collection of all possible sequences, at the level of only the inputs X and Y and the outputs Z, we can say that these two sets of algorithms can be viewed as computing exclusive-or. If we ask whether or not they retrieve their inputs in the same order, a different level of description is required. Furthermore, we can ask how it is that exclusive-or is calculated in each case. The answer provided above may be sufficient, but we may want to ask, in the case of the first version, 'Exactly how is the comparison between X and Y carried out?' In the case of the second version, we may want to ask, 'Exactly how is the logical "and" function evaluated?' In some scenarios, such as the logic student required to program exclusive-or from other available logic operators, these questions do not arise. In other possible contexts, it is conceivable that these questions would be crucial. Perhaps the first program runs smoothly, but the second program apparently fails each time a logical 'and' is performed. Granted, the two scenarios reasonably are framed at different levels of abstraction, in terms of detail, as they should be. But both are concerned with manipulating and understanding a real system, so it should not be said that the perspective of one is correct and essential while the perspective of the other is concerned with mere detail.

Comparison of Algorithms Represented by the PASCAL Programs and Turing Machine

It is very natural to consider the Turing machine algorithm in terms of states. For the case where the inputs are first a '0' (from left to right on the tape) and then a '1', the algorithmic sequence can be given as:

```
Q-STATE: q1    TAPE: 0

Q-STATE: q2    TAPE: 1

Q-STATE: q4    TAPE: B

Q-STATE: q7    TAPE: 0

Q-STATE: q9    TAPE: 1
```

A more detailed level of description might be that of the following:

```
Q-STATE: q1    TAPE: 01
TAPE POSITION: 1

Q-STATE: q2    TAPE: 01
TAPE POSITION: 2

Q-STATE: q4    TAPE: 0B
TAPE POSITION: 2

Q-STATE: q7    TAPE 0B
TAPE POSITION: 1

Q-STATE: q9    TAPE: 1B
TAPE POSITION: 1
```

This sequence includes a wider view of the tape, better capturing the left and right operations. Again, we are assuming that this is a correct description of a physical system.

In order to compare such a sequence with the PASCAL generated versions, we must find a common denominator, i.e., a common ideal machine for describing them both. This was accomplished without much ado for the two PASCAL versions. The two had identical variable names and the algorithmic sequences were chosen to emphasise their similarity at the less detailed level; at a certain level of detail from a certain perspective they fit easily into the same labels and values.

In order to find a common level of comparison for this more difficult case, let us start with the fairly high level view where the two PASCAL versions merge, the algorithmic sequence given by:

X: U Y: U Z: U

X: 0 Y: U Z: U

X: 0 Y: 1 Z: U

X: 0 Y: 1 Z: 1

Note that X, Y and Z are listed separately here, compared to the combined tape above. Starting from that Turing-machine-based sequence, we can abstract away to look at only the tape, and then go down a level or two to consider X, Y and Z separately, where X is the first square of the tape, Y is the second, and (interestingly) Z is again the first square. We get the algorithmic sequence shown by:

X: U Y: U Z: U

X: 0 Y: U Z: U

X: 0 Y: 1 Z: U

X: 0 Y: 1 Z: 1

Juxtaposed with the sequence generated from the PASCAL versions at a similar level of description (given below), several points can be made.

X: 0 Y: 1 Z: 0

X: 0 Y: 1 Z: 0

X: 0 Y: B Z: 0

X: 0 Y: B Z: 0

X: 1 Y: B Z: 1

First, there is no sequencing of the retrieval of input for the Turing machine; it is just there to begin with. Of course the PASCAL program could have been a procedure and could have had similar given input (this was not done because the ordering of inputs will make a useful comparison in the next section). So we have a difference in the input. We also have a potential difference in the values taken on, especially if 'U' is filling in for arbitrary values, although there may be a standard 'undefined' symbol, such as hexadecimal F's (all binary ones) in IBM machines.

Next, abstract from the PASCAL version by selecting the last two steps. Compare this to the first and last states of the Turing machine algorithm:

PASCAL versions:

X: 0 Y: 1 Z: U

X: 0 Y: 1 Z: 1

Turing machine version:

X: 0 Y: 1 Z: 0

X: 1 Y: B Z: 1

In the Turing machine sequence we see the state where the input is given, followed by the state at which the answer is reached. The interesting difference is that the Turing machine algorithm is what we might call 'destructive', destroying the input representations in the process of computing the answer. Finding a common level of description in terms of states makes it easy to make such comparisons even across such different 'languages' as PASCAL and Turing machines. Although, as is widely known, Turing machines and PASCAL-running VAXes are theoretically equivalent mathematically, it should be remembered that the types of effective procedures suited to them or natural to them are rather different. This point is not new, although Turing machines and von Neumann machines are considered similar, even equivalent, in the psychological literature. In any case, difficulties arise when claims are made about the suitability of particular programs to particular hardware devices. There is a grain of truth in this distinction, and this approach to comparison of algorithms in terms of different levels of ideal machine state sequences is an attempt to isolate and build on that grain of truth.

Comparison of Connectionist and Other Algorithms

In Figure 3.5, the connectionist diagram of Figure 3.3 is repeated, this time with numbers on the nodes for reference. One way of looking at this as representing an algorithm is to take the output values of each node as the values to watch. For inputs zero and one (input to nodes 0 and 1, coincidentally), we might get either of the following algorithmic sequences.

```
0: U      1: U      2: U
3: U      4: U

0: 0      1: U      2: U
3: U      4: U

0: 0      1: 1      2: U
3: U      4: U

0: 0      1: 1      2: 0.98
3: 0.02   4: 0.98

0: 0      1: 1      2: 0.98
3: 0.01   4: 0.99
```

```
0: U      1: U      2: U
3: U      4: U

0: U      1: 1      2: U
3: U      4: U

0: 0      1: 1      2: U
3: U      4: U
```

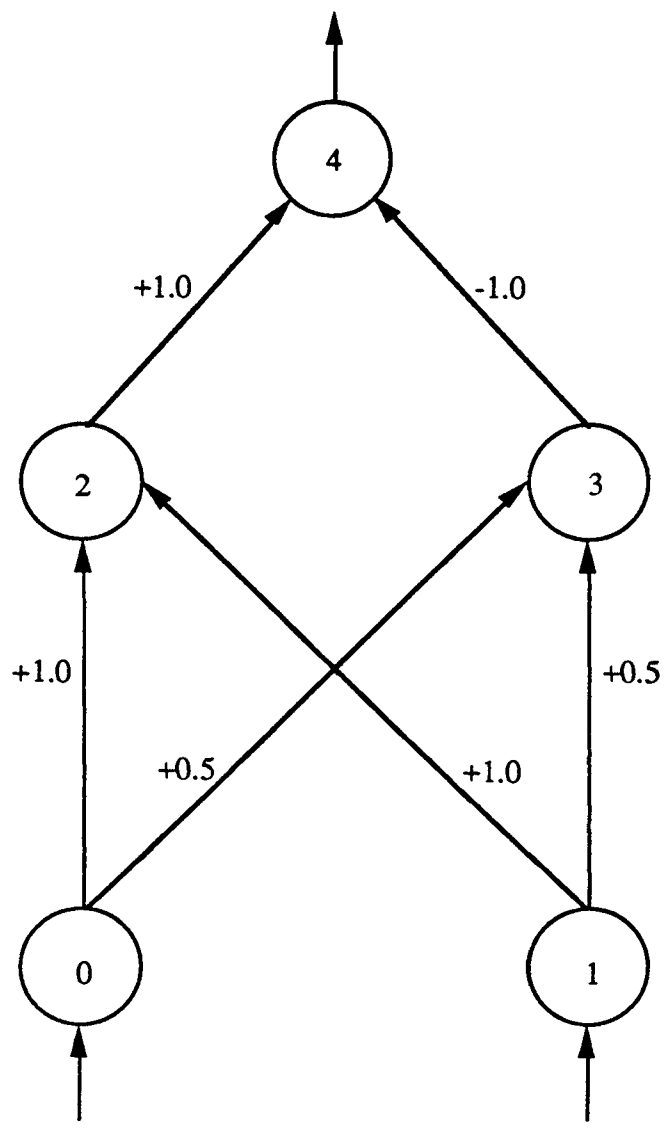

FIGURE 3.5

0: 0 1: 1 2: 0.97
3: 0.03 4: 0.97

0: 0 1: 1 2: 0.98
3: 0.01 4: 0.98

As a general comparison with the earlier algorithms, I will mention three main points. These are rounding, nondeterminism and sequencing. I have chosen to represent the output values as they are often represented in connectionist simulator programs. They could have been rounded to give only zeroes and ones, but then it would be impossible to see the development of the strength of the result. While it is not a very gradual development in this example, it can be in many connectionist models.

One might imagine that many different instances (runs) of a single physical system calculating exclusive-or and describable in this way will have slightly different values. In other words there is a certain amount of nondeterminism in the path from the starting point to the finishing point for this system for computing exclusive-or described at this level. That is, a state is not uniquely determined by the state preceding it, as in the deterministic PASCAL-inspired algorithms. If state changes are recorded from connectionist systems each time an output value changes by a set amount, the set of all possible sequences will most likely define a nondeterministic algorithm or ideal machine. This is not a problem, but perhaps it is a contribution to the difficulty of comparing connectionist and classical theories. While this real or simulated 'random' element clearly violates some definitions of effective procedures (as we have seen -- see the definitions of Enderton and Knuth for example), it is consistent with others, in particular Rumelhart and McClelland's usage

(as in their debate with Broadbent reviewed in the last chapter). In fact, deterministic Turing machines and nondeterministic Turing machines turn out to be (weakly) equivalent. (For a proof of this, see Davis and Weyuker, e.g.) In the new framework it can be seen that whether a system is determinstic or nondeterministic depends on the level of description. In this case, it may turn out that if we knew more about the underlying system and more about the initial conditions for a particular run, a level of description might be possible which includes label-value pairs for which any state could be distinguished deterministically from the previous state. To illustrate this point more simply, consider what we have been calling the top-level view of the exclusive-or function, as in the set containing the following four sequences:

X: 0 Y: 1 Z: U

X: 0 Y: 1 Z: 1

X: 1 Y: 0 Z: U

X: 1 Y: 0 Z: 1

X: 0 Y: 0 Z: U

X: 0 Y: 0 Z: 0

X: 1 Y: 1 Z: U

X: 1 Y: 1 Z: 0

This perfectly deterministic algorithm (for once all the sequences have been included) can be abstracted (by selection of values) to the nondeterministic algorithm comprising these four sequences:

X: 0 Z: U

X: 0 Z: 1

X: 1 Z: U

X: 1 Z: 1

X: 0 Z: U

X: 0 Z: U

X: 1 Z: U

X: 1 Z: 0

Whereas rounding could perhaps be used to find a common level of comparison with the earlier algorithms, by covering up the (not very) gradual approach to a solution, there is an arguably deeper difference. This is the order of the input. If it is true that at this level, the input can come into the system in any order, then this is different to the clear sequencing of the inputs in the other descriptions. This is not so surprising for a system typically described as parallel. In our new way of describing algorithms, this parallelness shows up as an indeterminacy in the path from inputs to outputs.

Given these differences, the search for a common level of comparison should start with rounding to zeroes and ones and also with selection of steps, leaving out the indeterminate steps to get the input in the first place. This gives us the following algorithmic sequence.

```
0: 0      1: 1      2: U
3: U      4: U

0: 0      1: 1      2: 1
3: 0      4: U

0: 0      1: 1      2: 1
3: 0      4: 1

0: 0      1: 1      2: 1
3: 0      4: 1
```

It could be generalised for all the possible sequences. A mere change of labels reveals that the first connectionist version at this level of description is surprisingly similar to one of the earlier sequences we have seen. It is a reasonable description of both systems to say that they compute 'X XOR Y' by first computing 'X OR Y' and 'X AND Y'. The two sequences do not yet line up exactly. This could be achieved by abstracting with selection of steps from the figure by omitting the ultimate (or penultimate) step. The second PASCAL version would require selection of the appropriate values, in this case X2, Y2, Z2, (X2 OR Y2) and (X2 AND Y2), and changing the notation, substituting '1', '0', 'X', 'Y' and 'Z' for 'TRUE', 'FALSE', 'X2', 'Y2' and 'Z2', respectively.

Similarly, the second connectionist model can be redrawn with node labels as in Figure 3.6, or as a sequence such as:

```
0: U      1: U      2: U
3: U      4: U

0: U      1: 0.99   2: U
3: U      4: U

0: 0.02   1: 1      2: U
3: U      4: U

0: 0      1: 1      2: 0.02
3: 0.97   4: U

0: 0      1: 1      2: 0.01
3: 0.99   4: 0.98
```

Changing the labels once again we have:

```
X: U      Y: U      X > Y: U
Y > X: U  NOT (Y = X): U

X: U      Y: 1      X > Y: U
Y > X: U  NOT (Y = X): U

X: 0      Y: 1      X > Y: U
Y > X: U  NOT (Y = X): U

X: 0      Y: 1      X > Y: 0
Y > X: 1  NOT (Y = X): U

X: 0      Y: 1      X > Y: 0
Y > X: 1  NOT (Y = X): 1
```

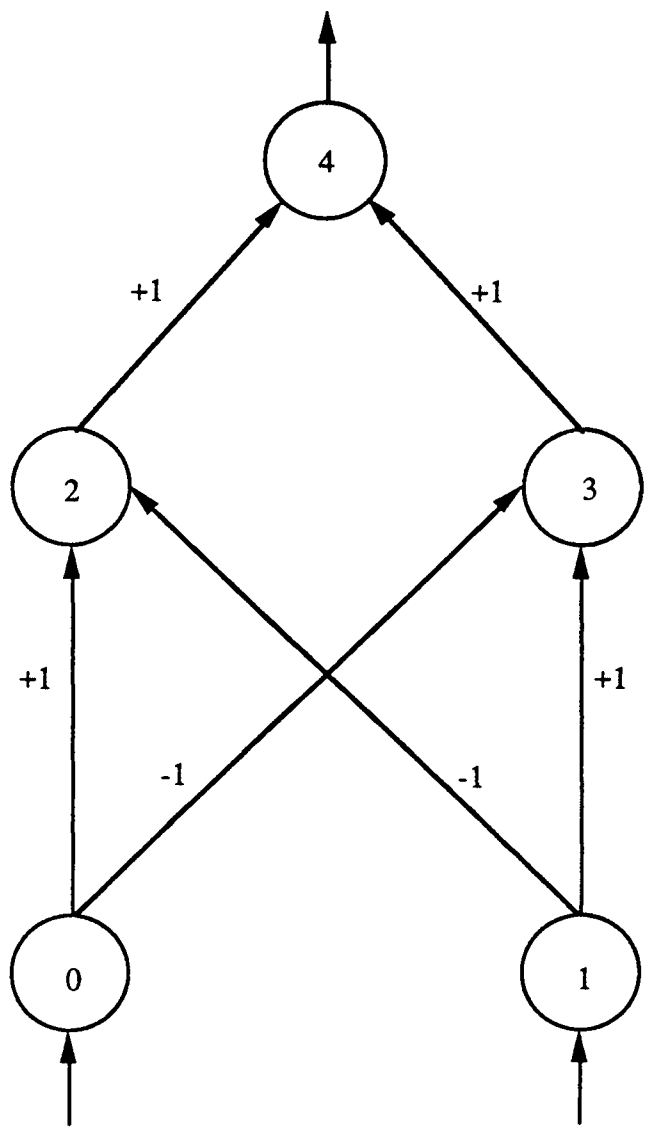

FIGURE 3.6

We can conclude that at this level of description, the algorithm might be equivalent to some PASCAL version using inequality checks -- at its appropriate level of description, of course.

In conclusion, the apparent similarity of the two PASCAL programs or that of the two connectionist systems is only skin deep. It is influenced by the striking similarity in notation as well as the implicit notion of states in each, something like theoretical computer science's states of a program in the former case and states of activation vectors in the latter. Under the old definition of algorithms as effective procedures, it is not even clear that the connectionist diagrams are algorithms at all. The new definition of algorithms in terms of sequences of more liberally defined states provides a framework for more meaningful comparisons based on strong equivalence at many levels of description. Under this definition, enhanced by abstraction operations, evidence can be given for grouping the equality-check PASCAL program with one connectionist system and the Boolean PASCAL program with the other, cutting across the classes based on superficial similarities.

CHAPTER IV
A MORE COMPLEX EXAMPLE: QUICKSORT

Before formalising the definitions of the last chapter, let us look at a more substantial example than exclusive-or. I have chosen Quicksort as that example for a number of reasons. It is a member of the family of sorting procedures (methods for putting sets of items in order) typically taught to and used by computer scientists. Because of this, there are a lot of text-book definitions which can be compared to each other and to sorting in general. In addition, there is a parallel flavour that is obscured by the usual renditions, but which I have come to think of as essential to Quicksort. It also has the right degree of complexity to be realistic, yet not so much that the main theoretical points will be overwhelmed. To avoid extraneous detail, only one set of inputs will be considered, with few exceptions. Therefore the basic algorithms will be singleton sets, containing only one sequence of states each. The arguments would contain no essential differences for generalised algorithms containing corresponding sequences for different input sets.

The remainder of the chapter will proceed as follows. An overview of Quicksort will be given first. Then, two standard sequential procedures drawn from textbooks will be compared, using techniques suggested by the definitions of the previous chapter. One of the procedures appears to be at a much higher level than the other, so it will be interesting to see if they are directly comparable at some level of description or not, and how this shows up in the new framework. A new high level version of Quicksort will then be given, one that suggests a parallel description. Finally, some more detailed differences will be noted between the original

two algorithms based on the textbook examples. The focus throughout is on comparison of algorithms, but it is also intended that the descriptive and explanatory role of algorithms at different levels will be elucidated along the way.

AN OVERVIEW OF QUICKSORT

Quicksort was introduced and described by C.A.R. Hoare (1961 and 1962). Like any conventional sort procedure, it takes as input a set of unordered, or rather incorrectly ordered, items, and produces the same set in a predefined order as its output. The example I shall concentrate on is taken from a first-year computer science text (Brookshear, 1988). The input is the sequence 'JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN'. The output under alphabetical ordering is of course 'ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM'. Perhaps due to the pervasiveness of von Neumann machines and the design of sorting procedures to fit them, the input is always given as ordered as well (though obviously not in the predefined sort order). Another constraint placed on Quicksort by this context is the desirability of sorting 'in place', using only the space required to store the original set of items plus a small fixed overhead for bookkeeping. From the outset, more detailed bottom-up concerns belie much of the alleged total autonomy of traditional algorithms. Hoare named his new sort 'Quicksort' because of the speed at which it can be executed on standard machines when the items to be sorted can fit into internal memory. It is sometimes called partition-exchange sort, for reasons which should become obvious.

What happens between input and output? One of the input items is chosen as a pivot. This choice can be made in a number of ways. Simply taking the first input item

will do. Using the pivot as the dividing point, the rest of the input is divided into two lists. The first list contains only items less than the pivot and the second list contains only items greater than the pivot. Some variation is possible with respect to those items exactly equal to the pivot. The pivot itself is not in either list, but goes in between them in the final ordering. By recursively applying Quicksort to the two new lists, we get an ordered list, followed by the pivot (in order), followed by another ordered list, completing the function. The recursion is stopped by the trivial cases of lists of zero items or one item only, which are already in order. Many variations on this are possible, such as randomly choosing the pivot or sorting small lists (less than some constant length) by some other method more suited to short lists.

The description thus far anticipates some aspects of my more parallel version of Quicksort in its lack of reliance on sublist ordering. However, as noted above, an essential aspect of traditional Quicksort is the clever way that the list is segmented and recombined in place. Using the example with JANE as pivot, two place markers are introduced. One starts at the beginning of the list and the other starts at the end. The beginning marker is advanced toward the end, one item at a time, moving along until the item marked is greater in value than the pivot (or until it reaches the end marker). In the example, the marker stops changing when it is pointing to 'TOM'. Similarly, the end marker is moved towards the beginning until an item is encountered that has a value less than the pivot (or the beginning marker is reached). After the markers have both been moved in this manner, either they are pointing to the same item or they are not. In the example, they are not:

```
JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN
(pivot)        *                        *
```

Here, the two marked items 'TOM' and 'CHERYL' are exchanged and the markers left in place. This ensures that everything before (and including) the first pointer is less than the pivot and everything after (and including) the second pointer is greater than the pivot. The process is repeated until the two pointers mark the same spot. Only one more pass is needed for the example, giving:

```
JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN
(pivot)                          **
```

Because the first sublist is less than the pivot, 'GEORGE' belongs with that sublist. Another clever aspect of the traditional sorting-in-place procedure is that the (doubly) marked item can now be exchanged with the pivot. Applying this action to the situation above has the effect of moving 'GEORGE' to the correct sublist and placing 'JANE' in the correct position for final output, partitioning the original list into two sublists to be sorted recursively in the same way:

```
GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN
(          new sublist                )    (new sublist)
```

Let us now take a look at two more precise formulations.

BROOKSHEAR'S QUICKSORT

Brookshear (p. 169) gives the following effective procedure in pseudocode. (The list is viewed as vertical with the beginning at the top.) The ranges of if, then, else, and while are indicated by indentation and parentheses.

```
procedure sort
if (the list contains fewer than two entries)
     then (declare the list sorted)
     else (select the first entry in the list
               as the pivot,
          place pointers at the first and last
               entries of the list,
          while(the pointers do not coincide)
               do   (move the bottom pointer up
                    to the nearest entry less
                    than or equal to the pivot
                    but not beyond the top
                    pointer,
                    move the top pointer down
                    to the nearest entry
                    greater than the pivot but
                    not beyond the bottom
                    pointer,
               if   (the pointers do not
                    coincide)
                    then (interchange the names
                         indicated by the
                         pointers)),
          interchange the pivot entry with the
               entry indicated by the common
               pointers,
          sort (the portion of the list above
               the pivot),
          sort (the portion of the list below
               the pivot))
```

An appropriate level of description to convey this information applied to the example input can be given using states, each of which consists of an ordered list of names, along with indices for specifying the pivot and top and bottom markers. There is some leeway here; for example the names themselves could be used for the pivot and markers. I have chosen to use the name for the pivot and integral indices for the markers. This reflects the procedure's description of top and bottom 'pointers' as opposed to list and pivot 'entries'. Labels correspond in general to pseudocode nouns following definite articles, though list items are grouped together. The 'WHOLE LIST' is also included to keep track of the whole list for this run in addition to the list being sorted by the current version of the

procedure. The multiple views of the list, assuming there may be just one such list in a particular realisation, brings in an abstraction operation not yet described. This is 'duplication'; it simply allows viewing the same lower level description in two ways at a higher level. Complete details are given in Chapter V.

Returning to the example, the initial state can be given as:

LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT: U TOP: U BOTTOM: U
SUBLIST1: U
SUBLIST2: U
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

The next state gives the situation after selecting the first name as the pivot and placing pointers at the top and bottom. (Changed values are underlined for legibility; this is not meant to be part of the state.)

LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT: JANE TOP: 1 BOTTOM: 10
SUBLIST1: ( )
SUBLIST2:
(BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

Since the top and bottom pointers are not coincident, they are moved in sequence, bottom pointer first, giving:

```
LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT:  JANE   TOP: 1   BOTTOM: 8
SUBLIST1: ( )
SUBLIST 2:
(BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT:  JANE   TOP: 4   BOTTOM: 8
SUBLIST1: ( )
SUBLIST 2:
(BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
```

Since TOP and BOTTOM are different, an interchange is carried out between the two names indicated by them:

```
LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
PIVOT:  JANE   TOP: 4   BOTTOM: 8
SUBLIST1: ( )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
```

Already, the kind of detail masked by this level of description can be seen. Exactly how comparisons are performed and even exactly which comparisons are performed are not shown. The former is not given by Brookshear either; the latter is implied in his discussion of the procedure but not given explicitly in the pseudocode. It is not clear from the discussion or

pseudocode whether the top pointer should start at the pivot or after it, something which the state descriptions at this level require and show precisely.

The top and bottom pointers do not yet coincide, so the while loop is repeated, again shifting the bottom and then the top marker:

```
LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
PIVOT:  JANE   TOP:  4  BOTTOM:  7
SUBLIST1: ( )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
PIVOT:  JANE   TOP:  7   BOTTOM:  7
SUBLIST1: ( )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
```

The end of loop condition has been satisfied, so the pivot is placed in its final position.

```
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
PIVOT:  JANE   TOP:  7   BOTTOM:  7
SUBLIST1:  (GEORGE BOB ALICE CHERYL CAROL BILL)
SUBLIST2:  (TOM SUE JOHN)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
```

Now all that remains is to sort the two new lists. According to Brookshear's procedure, the first sublist is sorted first. The following states provide a description of the process.

Initial state:

```
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT : U  TOP : U   BOTTOM : U
SUBLIST1 : U
SUBLIST2 : U
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
```

Set the pivot and pointers:

```
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT:  GEORGE   TOP:  1   BOTTOM:  6
SUBLIST1: ( )
SUBLIST2: (BOB ALICE CHERYL CAROL BILL)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
```

Move the bottom pointer (no change is made in the state), and then move the top pointer:

```
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT:  GEORGE   TOP:  6   BOTTOM:  6
SUBLIST1: ( )
SUBLIST2:  (BOB ALICE CHERYL CAROL BILL)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
```

```
Exchange the pivot ('GEORGE') with the item pointed to by
both pointers ('BILL'):

LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE)
PIVOT:  GEORGE  TOP:  6   BOTTOM:  6
SUBLIST1: (BILL BOB ALICE CHERYL CAROL)
SUBLIST2: ( )
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Next sort the new first sublist, starting by initialising
the pivot and pointers:

LIST:
(BILL BOB ALICE CHERYL CAROL)
PIVOT:  BILL   TOP  1   BOTTOM:  5
SUBLIST1:  ( )
SUBLIST2:  (BOB ALICE CHERYL CAROL)
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Then move the bottom pointer:

LIST:
(BILL BOB ALICE CHERYL CAROL)
PIVOT:  BILL   TOP:  1   BOTTOM:  3
SUBLIST1:  ( )
SUBLIST2: (BOB ALICE CHERYL CAROL)
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

The top pointer is shifted one place to BOB:

```
LIST:
(BILL BOB ALICE CHERYL CAROL)
PIVOT:  BILL   TOP:  2   BOTTOM:  3
SUBLIST1:  ( )
SUBLIST2:  (BOB ALICE CHERYL CAROL)
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

An exchange is carried out:

```
LIST:
(BILL ALICE BOB CHERYL CAROL)
PIVOT:  BILL   TOP:  2   BOTTOM:  3
SUBLIST1: ( )
SUBLIST2:  (ALICE BOB CHERYL CAROL)
WHOLE LIST:
(BILL ALICE BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

Move the bottom pointer:

```
LIST:
(BILL ALICE BOB CHERYL CAROL)
PIVOT:  BILL   TOP:  2   BOTTOM:  2
SUBLIST1:  ( )
SUBLIST2:  (ALICE BOB CHERYL CAROL)
WHOLE LIST:
(BILL ALICE BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

The top pointer remains the same; an exchange takes place between the pivot and the item pointed to by both pointers:

```
LIST:
(ALICE BILL BOB CHERYL CAROL)
PIVOT:  BILL   TOP:  2   BOTTOM:  2
SUBLIST1:  (ALICE)
SUBLIST2:  (BOB CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

Since the new first list consists of just 'ALICE' and is less than two items long, it is considered to be sorted already. Attention turns to the second sublist. The next state shows the values after initialisation of the pivot and pointers for this list:

```
LIST:
(BOB CHERYL CAROL)
PIVOT:  BOB  TOP:  1   BOTTOM:  3
SUBLIST1: ( )
SUBLIST2: (CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

This list '(BOB CHERYL CAROL)' is handled similarly, and so is the sublist at the end '(TOM SUE JOHN)'. A more detailed version of this example is given in Appendix A. The same type of states are used, but it also includes states reflecting the 'return' from each recursive call to an earlier state. The sequence of states included in this section form a legal abstraction of the algorithm given in Appendix A.

In addition to establishing a basis for comparison, this section provided an introduction to Quicksort at a typical level of description. In fact, Quicksort is often introduced to students with informal state diagrams showing the items to be sorted, the pivot and two pointers, followed later by more or less formal

procedural definitions.

KNUTH'S QUICKSORT

Now let us turn to a much more detailed and precise formulation of Quicksort from Volume 3 of Donald Knuth's The Art of Computer Programming (1973, pages 117 - 119). It is presented in MIX assembly language as defined by Knuth in Volume 1 (1968). I shall include it here, almost in its entirety, omitting only the timing information for each line and the final straight insertion sort intended for small lists. The comments include references to steps (Q1, Q2, etc.) and variables ('l' for left, 'r' for right, 'i' and 'j' for pointers -- much like TOP and BOTTOM in Brookshear's version). These and the MIX registers and storage locations are described in Appendix B, which also contains the detailed algorithm or state sequence for the example using input 'JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN'. It is not necessary to understand this somewhat daunting chunk of code completely. The relevant points should become clear in the discussion afterwards. Unless the reader is familiar with assembler languages, I suggest reading only the description in the appendix and the comments at the right in the procedure below, at least on the first time through. If even more details are required than are provided here, they can be found in Volumes 1 and 3 of Knuth's series.

Without further ado, the program:

```
A       EQU  2:3        First component of stack entry
B       EQU  4:5        Second component of stack entry
START   ENT6 0          Q1. Initialize.Set stack empty.
        ENT2 1          l <- 1.
        ENT3 N          r <- N.
2H      ENT5 1,3        Q2. Begin new stage.j <- r + 1.
        LDA  INPUT,2    K <- K1, R <- R1
        ENT4 1,2        i <- l + 1.
        JMP  0F         To Q3 omitting 'i <- i + 1'.
6H      LDX  INPUT,4    Q6. Exchange.
        ENT1 INPUT,4
        MOVE INPUT,5
        STX  INPUT,5    Ri <-> Rj
3H      INC4 1          Q3. Compare Ki:K. i <- i + 1.
0H      CMPA INPUT,4
        JG   3B         Repeat if K > Ki
4H      DEC5 1          Q4. Compare K : Kj.j <- j - 1.
        CMPA INPUT,5
        JL   4B         Repeat if K < Kj.
5H      ENTX 0,5        Q5. Test i : j.
        DECX 0,4
        JXP  6B         To Q6 if j > i.
        LDX  INPUT,5
        STX  INPUT,2    R1 <- Rj.
        STA  INPUT,5    Rj <- R.
7H      ENT4 0,3        Q7. Put on stack.
        DEC4 M,5        rI4 <- r - j - M
        ENT1 0,5
        DEC1 M,2        rI1 <- j - l - M.
        ENTA 0,4
        DECA 0,1
        JANN 1F         Jump if r - j ≥ j - l.
        J1NP 8F         To Q8 if M ≥ j - l > r - j.
        J4NP 3F         Jump if j - l > M ≥ r - j.
        INC6 1          (Now j - l > r - j > M.)
        ST2  STACK,6(A)
        ENTA -1,5
        STA  STACK,6(B) (l, j - 1) => stack.
4H      ENT2 1,5        l <- j + 1.
        JMP  2B         To Q2.
1H      J4NP 8F         To Q8 if M ≥ r - j ≥ j - l.
        J1NP 4B         Jump if r - j > M ≥ j - l.
        INC6 1          (Now r - j ≥ j - l > M.)
        ST3  STACK,6(B)
        ENTA 1,5
        STA  STACK,6(A) (j + l, r) => stack.
3H      ENT3 -1,5       r <- j - 1.
        JMP  2B         To Q2.
8H      LD2  STACK,6(A) Q8. Take off stack.
        LD3  STACK,6(B)
        DEC6 1          (l, r) <= stack.
        J6NN 2B         To Q2 if stack wasn't empty.
```

FINDING A COMMON ABSTRACTION

How does one begin to compare two such disparate procedures? First, what is alike about them? The framework introduced in this thesis allows a more formal statement of the question: Is there an abstract algorithm that can be implemented by either of the more detailed algorithms based on Brookshear's effective procedure and Knuth's program for Quicksort? What would it look like? Of course a trivial common abstraction is the one that describes sorting the example input by any means. The first state contains just the input and the second and final state contains just the output. Is there some algorithm at a lower level of detail that will still be an abstraction of both given algorithms?

As a starting point, we can try to describe the Knuth-based algorithm in the terms of the Brookshear-based one. While it has a shorter sequence, there is a sense in which the former algorithm seems to be a more detailed description, including details of comparisons and stacks (see the next section) for example. In Appendix C, we have a valid abstraction of the Knuth-based algorithm using the LIST, PIVOT, TOP, BOTTOM and SUBLIST labels inspired by Brookshear. (The particular abstraction operations are detailed in that appendix.) A common abstraction can be constituted by selecting the states wherein this description of the WHOLE LIST coincides with the WHOLE LIST of the Brookshear algorithm in Appendix A. Selecting the first states containing each new WHOLE LIST results in the following algorithm. (Again, value changes are underlined for clarity. Only the label-value pair for the WHOLE LIST is given, as all of the other values diverge for one or more of the selected states.)

```
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)

WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)
```

This new sequence says nothing about how the two sublists separated by 'JANE', in the third state above, are sorted to produce the final ordering of the fourth and last state. In addition, it says nothing about which sublist is sorted first or indeed if they are sorted in sequence or in parallel. In the constraints of the framework, of course, any lower level description is given in terms of definite sequences of states, but these may reflect simultaneous (at a given level of description) or interleaved changes to both lists. Nor does the above algorithmic sequence say anything about how the names to be exchanged are found; the details of the pointer movements have been suppressed.

To put this back into a procedural description, the algorithm including the above sequence could be abbreviated by the following pseudocode procedure, where states contain only the whole list and are changed whenever that list changes.

```
procedure newsort
     if (the list contains fewer than two entries)
          then
               (declare the list sorted)
          else
               ((select the first entry in the list as
               the pivot),
               repeat
                    (find the first entry in the list,
                    starting from the beginning, that
                    is greater than the pivot and find
                    the first entry in the list, starting
                    from the end, that is less than or
                    equal to the pivot)
                    if   (both searches are successful
                         and the found items are out of
                         order)
                         then (interchange them)
               until (only one search succeeds or both
                    succeed, with the found items in
                    order),
               interchange the pivot entry with the
                    entry found, if any, that is less
                    than or equal to it,
               newsort (the portion of the list above
                    the pivot)
          and  newsort (the portion of the list below
                    the pivot))
```

The use of 'and' above is intended to indicate that parallel processing is allowable. The 'repeat ... until' construct is a loop similar to 'while' except that the test (in this case that 'only one search succeeds or both succeed, with the found items in order') is not carried out until the body of the loop has been executed at least once. Newsort captures what is common to the two textbook Quicksort procedures and covers other

variations as well. The search for interchangeable entries can be carried out in many different ways, possibly even in parallel. The two sublists may be sorted in parallel, in the first-then-second order (as suggested by Brookshear's procedure), in shorter-then-longer order (as suggested by Knuth's program), or longer-then-shorter or second-then-first and so on.

A further abstraction is possible by mapping the ordered lists to unordered sets. While Quicksort is defined in terms of ordered input and ordered output, a more general algorithm can be defined which requires only the output to be ordered. Using duplication, we can view the WHOLE LIST in our earlier four-state sequence as unordered input and ordered output, as in

INPUT:
{JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN}
OUTPUT:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

INPUT:
{JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN}
OUTPUT:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

INPUT:
{JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN}
OUTPUT:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)

INPUT:
{JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN}
OUTPUT:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

This suggests an interesting variation on Quicksort that

could be realised physically with balls in urns for input (borrowing the illustrative device used in probability theory) and ordered output. Such an algorithm could be abbreviated by the following procedure. Note that we now have a random choice of pivot as in the original Quicksort formulation.

```
procedure newsort2
if   (the input contains fewer than two entries)
     then
          if   (the input contains exactly one entry)
               then
                    (place that entry in the first output
                         position)
          endif
else
     ((select an entry from the input as the pivot),
     (divide the remaining entries into two sets
          depending on whether they are less than or
          equal to the pivot or greater than the pivot,
          keeping a count of how many entries are in the
          less-than-or-equal set),
     (place the pivot in the output position one place
          past the count of how many entries are in the
          less-than-or-equal set),
     (sort the two sets created above))
```

The 'division of remaining entries' could have a fast physical realisation describable as parallel. Imagine a device with chutes for each ball and a check that the entry (a value on each ball -- a label or weight perhaps) is greater than a variable value. The pivot is chosen and the device set to check for the pivot's value. The remaining balls are rolled down the chutes and shunted into the appropriate bins, at least one of which has a counter at its entrance.

It is not so obvious how to abstract away from the choice of the initial entry as pivot in the von-Neumann-inspired algorithms presented earlier, nor is it obvious how to abstract away from the variations possible with respect to handling of other entries of equal value to the pivot. The realisation above however, suggests an efficient alternative for multiple equal entries. At the algorithmic level, this could be described in terms of a third set. When the entries are divided, they are placed in either of three sets depending on whether they are greater than the pivot (as before), or strictly less than the pivot, or equal to it. A count need only be kept for the less-than set. Finally, the pivot and all entries in the equal set are placed in consecutive output positions starting at one past the count of entries in the less-than set.

By considering Quicksort as state sequences divorced from the implicit constraints of the usual realisations, we have arrived at a rather different version with a possible realisation that is describable as highly parallel. A procedural abbreviation of the new algorithm follows.

```
procedure newsort3
if   (the input contains fewer than two entries)
     then
          if   (the input contains exactly one entry)
               then
                    (place that entry in the first output
                    position)
          endif
else
     ((select an entry from the input as the pivot),
     (divide the remaining entries into three sets
          depending on whether they are less than,
          greater than, or equal to the pivot, keeping a
          count of how many entries are in the less-than
          set),
     (place the pivot and all entries in the equal
          set in consecutive output positions starting at
          one past the count of entries in the less-than
          set),
     (sort the two remaining sets created above))
```

A COMPARISON OF LOWER LEVEL ALGORITHMS BASED ON THE EFFECTIVE PROCEDURES OF BROOKSHEAR AND KNUTH

It should be clear by now that any two Quicksort algorithms are not likely to be the same, although they probably have a common abstraction that contains more than just input and output. Whether this is actually true for a pair of algorithms is an empirical question. There is a huge amount of variation possible at the levels of detail implied by the usual descriptions of effective procedures and programs.

Let us now take another look at the algorithms based on Knuth's MIX program and Brookshear's effective procedure. A quick glance reveals that the MIX program is in a sense the 'lower level' one, specifying more detail. It would

not be a surprise if it were suggested that the MIX program is one of many possible implementations of Brookshear's higher level effective procedure. Some might even say that Brookshear's version is at the algorithmic level and that Knuth's is at the hardware implementation level -- it is close to machine code after all, concerned with registers, storage addresses and the like.

In the new framework developed here, however, the Knuth version, when taken as an abbreviation for a sequence of states, is not obviously an implementation of Brookshear's version, at least when the states are taken to include a level of detail similar to that implied by their respective textbook descriptions. There is some flexibility in the choice of detail to include. I shall use the storage areas explicitly mentioned in both cases as a guide to determine which label-value pairs to put in each state. A new state is added to a sequence for each statement or step that potentially affects one of these label-value pairs. More detail is possible. As an example, for the MIX program, I could have added the special registers for comparisons and jumps (not specified in the program but described elsewhere by Knuth), or even the storage of the statements of the program itself in machine code, along with an indication of the next statement to be executed. (This information is also discernible from Knuth's specification of MIX.)

Before examining the differences, note that there are at least two ways in which the Knuth algorithm could provide an explanation of some higher level algorithms. Even at the relatively gross level of description shown earlier in this chapter, representing just the items to be sorted whenever one of them is changed, more detail could be given about exactly how an exchange is carried out. Take, e.g., the first two steps of a sequence which

begins:

WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

(This could be abstracted from the sequences in Appendix A or C, or many other possible sequences.) Now compare this to a possible abstraction as defined above of the corresponding Knuth sequence. It begins:

WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE CHERYL SUE JOHN)

WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

This tells us something about how the exchange is carried out, or at least prompts a hypothesis. The two exchanged items are not switched simultaneously; rather, 'CHERYL' is copied across first. Of course, there may be some nondeterminism allowing the possibility of different methods of exchange even for this same input. But looking only at this case, more details are provided than in the previous one. Indeed, if we go back to the program and view it as an abbreviation for such state sequences, there is no such nondeterminism: this case will always be handled in this way. We might also theorise that there must be a still more detailed description of this system, revealing that 'TOM' is stored away so as not to be lost before the final state above. An examination of the algorithm in Appendix B

shows that it is indeed saved away in a place represented by rX.

Another example of detail given by the Knuth algorithm that could provide an explanation for a higher level algorithm is its handling of some comparisons between two values. rI4 and rI5 are compared by copying the value in rI5 into rX and then subtracting the value in rI4 from the value in rX. Of course, the movement of values in the example in Appendix B and other cases only provides supporting evidence for this theory. Even stronger evidence would be provided by a more detailed algorithm which revealed the interaction of the machine code statements equivalent to ENTX 0,5 (in MIX assembly language, this means 'Enter into rX the value contained in rI5, adding 0 to it first') and DECX 0,4 (in MIX: 'Decrement the value in rX by the value in rI4, subtracting 0 from it first') with these values.

Other comparisons are not spelled out in so much detail in the appendix. The comparisons governed by the instruction CMPA and implicit in jump instructions, such as JANN 1F or J1NP 8F, are examples of these. In MIX, CMPA INPUT,5 means 'Compare the value in rA to the input value at a position determined by the value in rI5'. A comparison indicator is set to LESS, EQUAL or GREATER, and it is used to control some jumps or branches, such as JG for 'Jump if greater'. JANN 1F means 'Jump if the value in rA is not negative, moving control to the first instruction in the forward direction labelled by 1H'. J1NP 8F means 'Jump if the value in rI1 is not positive; go to the next instruction in the forward direction labelled by 8H'.

Returning to differences that illustrate contradictions between the algorithms in Appendices A and B, we can see that neither is a straightforward implementation, or

more detailed version, of the other. Four examples are described here, though there may be more.

The first is the order in which sublists are handled after an initial list has been divided by the choice of pivot. In the algorithm based on Brookshear, the first sublist (top to bottom in his terminology, left to right in Knuth's) is always dealt with first. The Knuth-based algorithm sorts the shorter sublist first. It also allows arbitrary choice of the minimal number of items in a sublist to be sorted by some other method. This is obscured in the sample sequence in Appendix B by setting M (the minimum number of list elements) to 1 for more direct comparison with the other algorithm.

Both algorithms contain some detail about the movement of pointers, or the values taken on by TOP, BOTTOM, rI4 and rI5. The Brookshear-based algorithm changes the value of BOTTOM before TOP, considering the items at the end of the list first. Directly opposed to this (and not just a more detailed version of the same algorithm from this perspective), the Knuth-based algorithm starts at the left (or beginning) of the list of items. The sets of values taken on by the pointers in each algorithm are different as well. In the Brookshear-based algorithms, the top pointer is never allowed to pass below the bottom one and vice versa. (Perhaps starting at the bottom is related to this.) This is not the case for the Knuth-based algorithm, where rI4 and rI5 can pass each other by (and do in the example given). Indeed, rI4 and rI5 can take on values pointing just outside the list of items to be sorted. These positions are filled in with very low and very high values (represented by -INF and +INF, respectively) and so are treated much like the other values.

Another interesting difference in the two algorithms is

their handling of the recursion inherent in Quicksort. The Brookshear-based algorithm leaves the stack (or whatever it is that explains the recursion) implicit. At the level given, even small sublists (of length one or zero) are treated individually (although the two null sublists at either end of a list of length one are not). Now in common computer science parlance it might be said that such inefficiencies (assuming a conventional von Neumann system realisation) are 'implementation details' and can be cleaned up when the sort is programmed, or may even be remedied by a clever compiler. It is worth emphasising that in the terminology introduced in the last chapter, however, such a modified version would constitute a different algorithm, and one that is not necessarily a strict implementation of the one abbreviated by Brookshear's effective procedure. Extra states can be added at a lower level; none can be taken out. At a higher level, on the other hand, a more abstract algorithm may omit the states related to the trivial lists of length one or zero.

The Knuth-based algorithm is an example of the more 'efficient' kind of recursion handling alluded to earlier. The stack states are given explicitly (see Appendix B), so it can be seen that not every sublist is stacked. Lists of length one or zero are not stacked. Neither is the next list to be processed. In the entire example in Appendix B, only one list '(GEORGE BOB ALICE CHERYL CAROL BILL)' is ever stacked. This contrasts with the fourteen lists which are stacked in the Brookshear-based algorithm. This efficiency is apparent of course only with respect to the relative time and space of the algorithmic level of concreteness. The Knuth-based algorithm could be realised in a physical system that took years to stack a list; in that case it would be much slower than the usual classroom realisation of the Brookshear-based algorithm.

CHAPTER V
ALGORITHMS FORMALLY REVISITED

DEFINITIONS

Having seen how the new definitions of algorithms might be employed, a much more rigorous formulation of the framework can be tackled. In order to lay a solid foundation, Davis and Weyuker's (1983) version of a standard formalism is used as a guide. The detailed treatment of the preliminaries, covering sets and n-tuples, functions, quantifiers and some basic proof techniques is omitted here, as the reader can refer to this source or any of the many texts which include these basic ideas. In addition, Davis and Weyuker's version of alphabet and string definitions will be used. Once again, most details are omitted here, but the fundamental definition is (p. 3):

> An alphabet is simply some finite nonempty set A of objects called symbols. An n-tuple of symbols of A is called a word or a string on A. Instead of writing a word as (a1, a2, a3,..., an) we write simply a1a2a3...an ... The set of all words in the alphabet A is written A* ... We do not distinguish between a symbol a in A and the word of length 1 consisting of that symbol.

Strings in A can be concatenated to produce other strings in A. So for a1 and a2 in A, a1 ¦ a2 = a1a2; this signifies the concatenation of a1 and a2. These definitions are useful in giving a formal foundation for nice mnemonic labels, such as the ones already used in the examples.

Theoretical computer science provides a formal way (or rather many equivalent formal ways) of abstracting away from the time and space limitations of physical computing devices. In contrast, the goal of this project is to

abstract in the different way, away from details, as defined in Chapters II and III (and further developed in this chapter). Since the motivation and focus is on algorithms as descriptions of existing or hypothetical physical systems, finite space and time are constraints that apply at all levels of description. While this prevents some of the elegant generalisations which ignore finite space and time, in many ways the mathematical formulation is more straightforward. There is no halting problem for these finite algorithms.

Algorithms

To begin, we define our alphabet A = {'A', 'a', 'B', 'b', 'C', 'c', ..., 'Z', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '¦'. '&', ',', ' '}. A label (as we have been using the term, and not as Davis and Weyuker use it) can therefore be defined simply as an element of A*.

For values we start by allowing integers. Through techniques such as Gödel numbering and formally defined strings, it has been shown that finite sequences of numbers and characters can be encoded and represented by single nonnegative integers. The number -3.975 could be represented by the Gödel number for the sequence (1, 3975, 3), for example, where 1 indicates the value is negative, and 3 indicates the number of digits to the right of the decimal (or whatever) point. For convenience and without loss of rigour values can be written as they have been in preceding chapters, using strings (as defined over an alphabet -- A will do for both labels and values, but a different set could of course be used for each), sequences of values and decimal points. Values that can only be specified with an infinite number of digits, such as the fully expanded decimal value of pi or 1/3, are not legal.

So we can say all labels are drawn from A* and all values are drawn from the set of nonnegative integers, which we can call N+ (from N, the set of natural numbers, plus {0}). While the sets of possible labels and possible values are themselves infinite, any particular algorithm will use only a finite subset of these.

A set of labels is defined to be a finite nonempty subset of A* A state over a set of labels L is a set of ordered pairs (l, v), where l is a label and v is a value, including exactly one such pair for each label contained in L. We can also write a pair (l, v) as l: v, as in earlier chapters. The string delimiters (the inverted commas in 'x', for example) can be omitted from the label if the context is clear. Cases in which the context may not be clear include labels which contain commas.

An algorithmic sequence over a set of labels L is defined to be a sequence or list (order is important) of one or more, but at most finitely many, states over L.

Finally, an algorithm over a set of labels L is a finite set of algorithmic sequences over L. The constraint of finiteness is again a result of considering only descriptions of real physical systems. Discrete values in such descriptions can never be arbitrarily large, nor can infinitely many variations of a value be discerned due to limitations of meters (for more on this, see Chapter VI).

Such an algorithm defines an ideal machine. It can be defined exactly as an algorithm is defined, as a set of sequences of states. Alternatively, it can be seen as a set of valid states along with (possibly nondeterministic and context-sensitive) legal transactions between them.

Abstraction

At the heart of the theory of strong equivalence of complex systems is abstraction. It is defined in three parts, for states over a set of labels, for algorithmic sequences of such states and for algorithms in general. Abstraction and implementation are inverses of each other, and the terms are used here to refer to abstraction and implementation in terms of detail, not in terms of concreteness.

Most of the abstraction operations or relations were introduced in Chapter III, though less formally. Two new ones are discussed here. The first is duplication which we saw in Chapter IV. This is the only operation whereby the more abstract algorithm is in some sense bigger than the implementation. In these cases, the same lower level value may be seen in different ways from a higher level, e.g. a compressed value of ⟨2, 3⟩ might be separated to 2 and 3 at a higher level. By allowing an abstraction on states which duplicates one label-value pair, different functions or views of the duplicated value can be taken at still higher levels. The second new abstraction operation is the combination of states, defined as part of abstraction of sequences. Two adjacent states can be combined by combining the corresponding values from each state. Like the other abstraction operations, these have been defined as simple relations; more complex abstractions can be defined by combining them.

A state M1 over a set of labels L1 is defined to be an abstraction of a state M2 over a (possibly distinct) set of labels L2 (and M2 an implementation of M1) if and only if one, or more, of the following cases holds.

(All functions in the cases are defined for valid states {(l1, v1), (l2, v2), ..., (ln, vn)}).

Case 1. This is 'selection of values' (or more precisely of label-value pairs) for an individual state, in the more procedural language of Chapter III.

M1 is a subset of M2.

In other words, M1 = fL(M2), where fL is a function defined as

fL({(l1, v1), (l2, v2), ..., (ln, vn)}) = {(li, vi), such that (li, vi) is in M2 and li is in L}, for some nonempty set L which is a subset of L2, the set of labels of M2.

For example, let M1 = {('x', 0), ('z', 0.5)} and let M2 = {('x', 0), ('y', 1), ('z', 0.5)}. Then M1 is an abstraction of M2, and M2 is an implementation of M1.
L = {'x', 'z'}.

Case 2. This is 'change of labels', or 'lateral abstraction'. Zero or more of the labels in M1 and M2 are different, while the values are exactly the same and remain paired with their corresponding labels.

More formally, M1 = fg(M2), where fg is a function defined as

fg({(l1, v1), (l2, v2), ..., (ln, vn)}) = {(g(l1), v1), (g(l2), v2), ..., (g(ln), vn)}, for some function g which defines a one-to-one correspondence between L2 and L1.

In an example similar to the one for Case 1, let M1 = {('x', 0), ('y', 1), ('x ¦ y', 0.5) and let M2 be the same as above, {('x', 0), ('y', 1), ('z', 0.5)}. Then M1 is an abstraction of M2 once again, and M2 is an

implementation of M1. M2 is also an abstraction of M1 by change of labels, and M1 an implementation of M2.

The function g maps labels of M2 to labels of M1 according to the correspondence:

'x' -> 'x'
'y' -> 'y'
'z' -> 'x ¦ y'

Similarly, g-1 maps labels of M1 to labels of M2 by the reverse correspondence:

'x' -> 'x'
'y' -> 'y'
'x ¦ y' -> 'z'

Case 3. This is 'generalised rounding', covering any transformation of individual values between states.

M1 = fg,lj(M2), where fg,lj is a function defined by

fg,lj({(l1, v1), (l2, v2), ..., (ln, vn)}) = the union of {(lj, g(vj))} and {(li, vi), li in L1 = L2, li not equal to lj}, for exactly one lj in L1 = L2, and for some function g from N+ into N+.

If M1 = {('x', 0), ('y', 1), ('z', 1)} and M2 = {('x', 0), ('y', 1), ('z', 0.5)}, then M1 is an abstraction of M2 (where g(0.5) = 1 and lj = 'z'), and M2 is an implementation of M1. M2 is also an abstraction of M1 by this case, where g(1) = 0.5 and lj = 'z'.

Case 4. This is 'grouping of values' (and corresponding grouping of labels).

M1 = flj,lk(M2); flj,lk is a function defined by

flj,lk({(l1, v1), (l2, v2), ..., (ln, vn)}) = the union of {(lj ¦ ',' ¦ lk, <vj, vk>)} and {(li, vi), li in L2 not equal to lj or lk)}, for exactly one lj and one lk in L2, such that lj does not equal lk.

An example is M1 = {('x, y', <0, 1>), ('z', 0.5)} and M2 = {('x', 0), ('y', 1), ('z', 0.5)}. M1 is an abstraction of M2, and M2 is an implementation of M1, where lj = 'x' and lk = 'y'.

Case 5. This is 'duplication', allowing the abstract algorithm to show one implementation value twice. The repeated value is paired with a label formed by appending a '2' to the end of the label of the original value.

M1 = flj(M2), where flj is a function defined by

flj({(l1, v1), (l2, v2), ..., (ln, vn)}) = the union of {(lj ¦ '2', vj)} and M2, for exactly one lj such that lj is in L2.

For example, if M1 = {('a', 1), ('a2', 1), ('b', 2)} and M2 = {('a', 1), ('b', 2)}, M1 is an abstraction of M2 by duplication. M2 is also an abstraction of M1, but by selection of values.

Case 6. This is 'chain of abstraction for states'.

There exist states Mi1, Mi2, ..., Min (n in N), such that M1 is an abstraction of Mi1 by one of Cases 1 - 5, Mi1 is an abstraction of Mi2 by one of Cases 1 - 5, ..., and Min is an abstraction of M2 by one of Cases 1 - 5. This allows arbitrarily (but finitely) many combinations of the first five cases.

M1 = f(M2), where f is the composition of the appropriate defining functions of Cases 1 - 5.

For example, let M1 = {('input', <0, 1>), ('output', 1)} and M2 = {('x', 0), ('y', 1), ('z', 0.5)}. That M1 is an abstraction of M2 can be shown in this way: Let Mi1 = {('x, y', <0, 1>), ('z', 1)}. M1 is an abstraction of Mi1 by Case 2 (change of labels). Let Mi2 = {('x', 0), ('y', 1), ('z', 1)}. Mi1 is an abstraction of Mi2 by Case 4, grouping of values and labels. Finally, Mi2 is an abstraction of M2 by Case 3 (generalised rounding).

The definition of abstraction from states to states forms the basis of abstraction from and to sequences and algorithms in general. The rather minimal constraints of state abstraction are strengthened by the requirement of uniformity of state abstraction across sequences on their own and within algorithms.

An algorithmic sequence S1 is defined to be an abstraction of an algorithmic sequence S2 (and S2 is an implementation of S1) if and only if one, or more, of the following holds:

Case 1. This is 'selection of states'.

S1 has strictly fewer states than S2, and any states occurring in S1 also occur in S2 and they occur in the same order. S2 is not an abstraction of S1 by this case; there is at least one state in S2 that is not in S1.

Case 2. This is 'uniform state-by-state abstraction'.

If S2 = (M1, M2, ..., Mn), then S1 = F(S2), for some function F defined by

F((M1, M2, ..., Mn)) = (f(M1), f(M2), ..., f(Mn)), for some function f which defines f(Mi) as a valid state abstraction of Mi, for each state Mi of S2.

Case 3. This is 'combination of states'.

S1 contains one less state than S2, and for exactly two adjacent states M2a and M2b of sequence S2, there exists state M1a in sequence S1 such that

M2a = {(l1, v12a), (l2, v22a), ..., (ln, vn2a)},
M2b = {(l1, v12b), (l2, v22b), ..., (ln, vn2b)}, and
M1a = {(li, <vi2a, vi2b>), i = 1, ..., n}.

All other corresponding states in S1 and S2 are equal, and they retain their order relative to each other and to the combining (or combined) states.

Case 4. This is 'chain of abstraction for algorithmic sequences'.

There exist algorithmic sequences Si1, Si2, ..., Sin, such that S1 is an abstraction of Si1 by Case 1, 2 or 3, Si1 is an abstraction of Si2 by Case 1, 2 or 3, ..., and Sin is an abstraction of S2 by Case 1, 2 or 3. This allows arbitrarily (but finitely) many combinations of the first three cases.

An algorithm A1 is defined to be an abstraction of an algorithm A2 (and A2 is an implementation of A1) if and only if either of the following holds:

Case 1. This is 'selection of sequences'.

A1 is a subset of A2.

Case 2. This is 'abstraction over sequences'.

A1 is not a subset of A2, but for every algorithmic sequence S1 in A1, S1 is an abstraction of S2 for some algorithmic sequence S2 in A2.

THEOREMS

A number of theorems follow directly from the definitions. A sampling of such theorems is included here with proofs to show how the formal multilevel theory of algorithms can be expanded and enriched, and to give the flavour of the proofs.

Theorem (trivial abstraction): Every algorithm, algorithmic sequence and state over a set of labels is an abstraction (and implementation) of itself.

Proof of the theorem: Working backwards from states to algorithms, let S be a state over a set of labels M. Then S is a subset of S, so S is an abstraction of itself, by Case 1 of the definition of state abstraction.

If Seq1 and Seq2 are algorithmic sequences and Seq1 = Seq2, then for every state S in Seq1, S is an abstraction of itself (from the portion of the proof just given above) and therefore of the corresponding state in Seq2. By uniform state-by-state abstraction, then, Seq1 is an abstraction of Seq2 = Seq1.

For algorithms, for every algorithm A, A is a subset of A, and therefore an abstraction of A (by Case 1 of the definition of algorithmic abstraction).

Theorem (another kind of trivial abstraction): The empty set is an abstraction of every algorithm.

Proof of the theorem: Again, this follows from Case 1 of algorithmic abstraction, since {} is a subset of A, where A is an arbitrary algorithm.

Theorem: The union of two algorithms defined over the same set of labels is again an algorithm, and the resultant new algorithm is an implementation of each of the two original algorithms. (The union of two states over the same set of labels is not a valid state at all, in general.)

Proof of the theorem: Let A1 and A2 be algorithms over some set of labels M. Then both A1 and A2 contain only sequences over M. Let A3 be the union of A1 and A2. A3 contains only sequences over M, since any one of its elements is also an element of either A1 or A2 (or both). Therefore, A3 is an algorithm over M. Since A1 and A2 are subsets of A3, they are abstractions of A3 and A3 is an implementation of each of them. (The parenthetical claim is proved with an example. Let M = {'x'} be a set of labels. Then {('x', 0)} is a state over M, as is {('x', 1)}. The union of these two states gives {('x', 0), ('x', 1)}, which is not a valid state over M: it is not in one-to-one correspondence with M and it has two pairs with the same first element, or label.)

Theorem: The empty set is not a valid state over any set of labels.

Proof of the theorem: Let S be a state over a set of labels M. Assume S is empty. By definition, S contains an ordered pair for each member of M, so M must be empty. But this contradicts the definition of a set of labels as a finite nonempty subset of A*. The assumption must have been false; S cannot be both empty and a state over M.

Theorem: Given any two algorithms A and B, there exists an algorithm C such that C is an abstraction of A and C is an abstraction of B.

Proof of the theorem: Since the empty set is an abstraction of all algorithms (including A and B), let C = {}.

This trivial abstraction is somewhat unsatisfying. We can prove something stronger as it turns out.

Theorem: Given any two nonempty algorithms A and B, there exists a nonempty algorithm C such that C is an abstraction of A and C is an abstraction of B.

This too turns out to be quite simple, once we prove the following lemma.

Lemma: {{{('x', 1)}}} is an abstraction of every nonempty algorithm.

Proof of the lemma: Let C1 be a nonempty algorithm. Choose an element Seq1 from C1. (C1 is an algorithm and therefore finite, so choice is uncontroversial. If more detail is desired, a rule can be specified, such as 'choose the pair with the first label, according to the numeric ordering of labels'.) Then C2 = {Seq1} is an abstraction of C1, because it is a subset of C1.

We can continue to 'abstract up' to {{{('x',1)}}}, by applying selection of states, selection of values, change of labels and rounding (the latter three as part of uniform state-by-state abstraction) to the sequence of C2.

Seq1 must be equal to (S1, S2, ..., Sn), where n>0 and each Si is a state over M. Then (S1) is an abstraction

of Seq1 (by selection of states), so C3 = {(S1)} is an abstraction of C2 and of C1 (by chain of abstraction).

Now, looking more closely at the state itself, S1 = {(l1, v1), (l2, v2), ..., (lm, vm)} for l1, l2, ..., lm in M and v1, v2, ..., vm in N+. By selection of values, we have S1' = {(l1, v1)}, and S1' is a subset of S1. Therefore, S1' is an abstraction of S1 and C4 = {({(l1, v1)})} is an abstraction of C3 and of C1.

C5 = {({('x', v1)})} is an abstraction of C4, by change of labels, using the correspondence 'x' <-> l1.

Finally, we can find a very gross rounding function g which maps all v in N+ onto 1, and in particular maps v1 onto 1. When this function is applied to the values for 'x' in the state in the sequence of C5, we get {({('x', 1)})} as an abstraction of C5, and therefore of C1, the arbitrary nonempty algorithm that was our starting point. This ends the proof of the lemma.

Proof of the theorem: Let C = {({('x', 1)})}. By the lemma, C is an abstraction of A and B, and the proof is complete.

An informal restatement of the two preceding theorems is that any two systems are equivalent at some level of abstraction. It is most likely possible to prove still more complex common abstractions exist for more complex cases.

Most importantly, the framework provides a mechanism for proving the equivalence of algorithms and for showing whether or not one is an abstraction of the other. In the next two sections we see that the framework is not too weak to be meaningful and that it provides the means

for proving a restricted (finite) version of Church's Thesis.

The following results will come in useful for the next section. First, a notational definition:

Definition: If S is an algorithmic sequence, then |S| is the number of states in S.

Theorem: Let S1 and S2 be algorithmic sequences such that S1 is an abstraction of S2. Then |S1| is less than or equal to |S2|.

Proof of the theorem: Let S1 have m states and S2 have n states. Consider the four cases of the sequential abstraction relationship.

Case 1. Selection of states.

In this case the n states of S1 all occur in S2, and at least one other state is in S2 as well, so $m < n$.

Case 2. Uniform state-by-state abstraction.

A precondition for this case is that $m = n$.

Case 3. Combination of states.

Here, S1 has one less state than S2, so $m = n - 1$, and $m < n$.

Case 4 allows any finite combination of the above. S1 is an abstraction of some sequence Si1, which is an abstraction of Si2, ..., which is an abstraction of S2. By the first three cases above, we can conclude that at each stage

$|S1| \leq |Si1| \leq |Si2| \leq \ldots \leq |Sin| \leq |S2|$.

Therefore, $|S1|$ is less than or equal to $|S2|$, which is what we set out to prove.

Corollary: If S1 is a sequential abstraction of S2 and $|S1| = |S2|$, then for any intermediate abstraction S, $|S| = |S1| = |S2|$.

Proof of the corollary: This is a simple consequence of the theorem, with an intermediate sequence of states being squeezed between the more and less abstract sequences.

More explicitly, we know from the theorem that $n \leq |S|$ and that $|S| \leq n$, where $n = |S1| = |S2|$. So $|S|$ must be n as well.

Theorem: Let S1 and S2 be algorithmic sequences, where $|S1| = |S2|$ and S1 is an abstraction of S2. Then the abstraction can be defined by Case 2 of the definition of sequential abstraction (uniform state-by-state abstraction).

Proof of the theorem: Consider each of the other cases in turn (if not in order) of the definition of sequential abstraction.

Case 1. For this case to hold, S1 must have strictly fewer states than S2, but this is not true. Furthermore, by the above corollary this sort of abstraction cannot occur in any chain of abstraction between S1 and S2, because $|S1| = |S2|$.

Case 3. Similarly, here S1 must have one less state than S2. The same argument holds as for Case 1.

Case 4. This is the combination case, allowing a finite chain of sequential abstractions. We have already seen however that no chain can include Case 1 or Case 3 abstraction, leaving only single or multiple instances of Case 2 (uniform state-by-state) abstraction. But it turns out that these are equivalent. To complete the proof, then, we prove the following lemma.

Lemma: Let S1 and S2 be algorithmic sequences such that S1 is an abstraction of S2 by Case 4 of the definition of sequential abstraction, comprising multiple instances of uniform state-by-state abstraction. Then S1 is an abstraction of S2 by just one instance of uniform state-by-state abstraction.

Proof of the lemma: By Case 4 of the definition of sequential abstraction (for multiple abstractions) and the assumptions of the lemma, there exist algorithmic sequences Si1, Si2, ..., Sin, such that S1 is an abstraction of Si1 by uniform state-by-state abstraction, Si1 is an abstraction of Si2 by uniform state-by-state abstraction, ..., and Sin is an abstraction of S2 by uniform state-by-state abstraction. By Case 2 of the same definition (for uniform state-by-state abstraction), if M1j is the jth state of S1, Mi1j is the jth state of Si1, etc., then M1j is a state abstraction of Mi1j, which is a state abstraction of Mi2j, ..., which is a state abstraction of M2j, for j from 1 to |S1|. But this fits the definition of state abstraction by Case 6 (finitely many multiple abstractions), so each M1j is an abstraction of each M2j, using the uniform functions, or subsets of them. So S1 is an abstraction of S2 by a single instance of uniform state-by-state abstraction. (This can be thought of as pushing the chain of abstraction back to the state description and away from the sequence description of abstraction.) This

completes the proofs of both the lemma and the preceding theorem.

MORE THEOREMS: NOT EVERYTHING IMPLEMENTS EVERYTHING

In terms of the framework built up throughout this thesis and, in particular, the definitions given in the previous section of this chapter, the theorem that not everything can be implemented by everything may be stated as follows.

Theorem. There exist algorithms A1 and A2 such that A1 is not an implementation of A2.

If this is true, then we also have the easy corollary which comes from simply reversing (the content of) A1 and A2:

Corollary. There exist algorithms A1 and A2 such that A1 is not an abstraction of A2.

Proof of the corollary. This follows from the definitions of implementation and abstraction, plus the theorem to be proved.

Proof of the theorem. To prove the theorem, all that is required is an example of such a pair of algorithms A1 and A2, along with a demonstration that the first cannot be viewed as an implementation of the second.

Let A1 = {}
Let A2 = {Seq2}, where Seq2 = ({'x', 0}).

A2 is not a subset of A1, nor is it true that Seq2 is an abstraction of some algorithmic sequence in A1. Therefore A2 is not an abstraction of A1 and A1 is not an implementation of A2.

Only slightly less straightforward is the proof for sequences.

Theorem. There exist algorithmic sequences Seq1 and Seq2 such that Seq1 is not an implementation of Seq2.

Proof of the theorem. Let Seq1 = (S1) and let Seq2 = (S2, S3), where S1, S2 and S3 are states.

If we assume Seq1 is an implementation of Seq2, then Seq2 is an abstraction of Seq1. By an earlier result we know that that implies ¦Seq2¦ is less than or equal to ¦Seq1¦. But that would mean 2 is less than or equal to 1, a contradiction.

It is more complicated to show the following:

Theorem. There exist algorithmic sequences Seq1 and Seq2 where ¦Seq1¦ = ¦Seq2¦ and Seq1 is not an implementation of Seq2.

Proof of the theorem.

Let Seq1 = ({('x', 0)}, {('x', 0)}).
Let Seq2 = ({('x', 0)}, {('x', 1)}).

The question of whether Seq2 is a legal abstraction of Seq1 reduces to the question of whether or not there exists a uniform state-by-state abstraction from Seq1 to Seq2. This reduction is a direct consequence of the final theorem of the earlier section, 'Theorems', of this chapter. Paraphrasing, that theorem states that two algorithmic sequences with the same number of states, one implementing the other, are related by one instance of uniform state-by-state abstraction.

What remains to be shown, then, can be given as a lemma:

Lemma. Seq2 is not a possible (sequential) abstraction of Seq1 by uniform state-by-state abstraction, where Seq1 = ({('x', 0)}, {('x', 0)}) and Seq2 = ({('x', 0)}, {('x', 1)}).

First, a more general result can be shown. Call it

The State-by-State Function Lemma: If S1 is a (sequential) abstraction of S2 then there exists a function F, such that if S1 = (M11, M12, ..., M1n) and S2 = (M21, M22, ..., M2n), then M11 = F(M21), M12 = F(M22), ..., M1n = F(M2n).

Proof of the State-by-State Function Lemma. This follows directly from the definitions of uniform state-by-state sequential abstraction and the six cases of state abstraction. The first five cases of state abstraction are defined by showing the function from the implementation state to the abstract state. The sixth case (a chain of abstractions of any of the first five sorts) gives the relation as a composition of functions, which is again a function. The uniformity constraint of the sequential abstraction definition ensures that the same function F applies for each M1i = F(M2i).

Proof of the first lemma. Seq2 = ({('x', 0)}, {('x', 1)}) and Seq1 = ({('x', 0)}, {('x', 0)}). By the State-by-State Function Lemma, there exists a function F such that F({('x', '0')}) = {('x', '0')} and F({('x', '0')}) = {('x', '1')}. So either F is not a function (a contradiction) or Seq2 is not an abstraction of Seq1 by uniform state-by-state abstraction. This completes the proof of the first lemma.

The proof of the main theorem is now complete as well.

ALGORITHMS AND COMPUTABILITY: WEAK EQUIVALENCE

The class of functions computed by algorithms is the same as that computed by Turing machines with finite tapes or programs with bounded input, and where the functions are defined for each of the finitely many finite inputs. Note that nothing is said about inputs that are not covered by the definition. We might call this class the class of bounded computable functions or the class of functions computed by finite tape automata. Finite tape automata can be distinguished from finite state automata in that the latter tend to be defined for an infinite set of inputs. In the theoretical computer science and philosophical literature, finite tape automata and finite state automata are often (erroneously) equated. Davis and Weyuker (1983, p. 149) contrast them to general Turing machines:

> At the opposite pole, one can imagine a device which moves from left to right on a finite input tape, and it is just such devices, the so-called finite automata, that we will now study. (My emphasis first, and then theirs).

Yet it is hard to find an example in their text of a finite language accepted by such a machine. More typical is the standard example of $a^n b^m$, where n and m are any integer and a and b are in the alphabet.

Putnam also equates finite tape machines and finite state machines (Putnam, 1967; p. 409 in Putnam, 1975), commenting parenthetically that

> Of the many useful equivalent definitions of 'finite automaton', the most useful for present purposes is the one that results if the definition of a Turing Machine is modified, by specifying that the tape should be finite.

Indeed Chomsky's famous demonstration (1957, p. 21) that 'English is not a finite state language' would be rather different (and trivial to prove, given that all his example input sets are infinite) if stated in terms of finite tape machines.

The functions computed by finite tape automata are not even a subset of the computable functions, strictly speaking, due to the latter being defined over infinitely many inputs. Somewhat surprisingly considering their finite restrictions, the class of functions described by algorithms is better seen as a subset of partially computable functions. To leave an output undefined in automata theory usually equates to the ideal system not halting for that input. But our notion of implementation leaves open a definition for other inputs at a lower level of detail. So even casting the class of functions computed or described by algorithms as a subset of partially computable functions does not capture the finite multilevel flavour of algorithms very well. What we have is a notion of 'undefined' such that for a given input value and a given algorithmic description of a system that leaves its output undefined, we cannot say what happens (whether the underlying system halts or not, e.g.) or even if it is possible to input that value into the system -- at that level of description.

PROVING STRONG EQUIVALENCE

The starting point for this section is Davis' version of Church's Thesis (from Davis and Weyuker, 1983), put in terms of his theoretical programming language S, though any suitably defined language will do (p. 54):

> ... we have reason to believe that any algorithm for computing on numbers can be carried out by a program of S ... The last italicized assertion is a form of what has come to be called Church's thesis.

They add, as noted in Chapter III,

> ... since the word 'algorithm' has no general definition separated from a particular language, Church's thesis cannot be proved as a mathematical theorem.

It is not necessary to know all the details of the formal definition of S; it is a procedural language which abstracts away from time and space. From Davis and Weyuker, pp. 15 - 16:

> We will use certain letters as variables whose values are numbers. (In this book the word number will always mean non-negative integer, unless the contrary is specifically stated.) In particular, the letters
>
> X1 X2 X3 ...
>
> will be called the input variables of S, the letter Y will be called the output variable of S, and the letters
>
> Z1 Z2 Z3 ...
>
> will be called the local variables of S ... Unlike the programming languages in actual use, there is no upper limit on the values these variables can assume ...
>
> In S we will be able to write 'instructions' of various sorts; a 'program' of S will then consist of a list (i.e., a finite sequence) of instructions.
>
> Output and local variables are initialised to 0, and there are three kinds of instructions, any of which can have a label. They are illustrated as (p. 16):

V <- V+1 Increase by 1 the value of the variable V.

V <- V-1 If the value of V is 0, leave it unchanged; otherwise decrease by 1 the value of the variable V.

IF V = 0 GOTO L If the value of V is nonzero, perform the instruction with label L next; otherwise proceed to the next instruction in the list.

More complex instructions, or macros, are based on these, and all can be formally defined. Macros are best seen as abbreviations; the numbers and computations (see below) are defined for the more basic underlying program.

Granted, the new definition of algorithms evolved up to this point is rather different to Davis' definition in terms of S or any other such language. Still, the two notions have a lot in common, and each has its uses (see Chapter III, 'Algorithms', for more detail). However algorithms in the new multilevel, state sequence framework are not included under the above proscription against mathematical theorems: what they provide is first and foremost a common framework, intended primarily to cut across boundaries of specific languages, machines or complex systems in general. The consequent evolved algorithm carries its own constraints; it is doubly finite, in time and in space. With this proviso, we can prove a strong equivalence theorem, like a very restricted version of Church's Thesis.

As a first iteration, consider the exact wording above: 'Any algorithm for computing on numbers can be carried out by a program of S'; look initially at 'Any algorithm for computing on numbers ...'. The changes from Church's Thesis are concealed in the changed definition of 'algorithm'. '... for computing on numbers ...' can stay; the new algorithms have numbers at their centres

too, and indeed they were modelled closely on the numeric foundations of S. '... a program of S ...' is completely defined and presents no theoretical problem, though the sheer amount of detail in its complete definition must be managed with care. That leaves '... can be carried out by ...' as in 'Any algorithm for computing on numbers can be carried out by a program of S.' For Davis, this means simply that we can write a program that computes the function as (weakly) defined by the algorithm. Not having or wanting a particular language, and with a stronger equivalence in mind, some adjustments are required.

What about 'Any algorithm (in the new sense) for computing on numbers is an abstraction of the set of computations* of a program of S', where computations* are defined as follows? Recall from Chapter III that a computation of a program is a list of snapshots in order of their occurrence. Each snapshot contains the line number of the instruction to be executed next, along with the current values of all variables in the program. There are a few minor niceties to be tidied up; line numbers are included in computations, and the label or variable-value pair notation is slightly different. The * is meant to signify these minor changes. This can be defined precisely as the mapping f from computations to computations*, where

```
f(((i1, {V1 = v11, V2 = v12, ..., Vm = v1m}),
   (i2, {V1 = v21, V2 = v22, ..., Vm = v2m}),
    .
    .
    .
   (in, {V1 = vn1, V2 = vn2, ..., Vm = vnm})))

= ({('V1', v11), ('V2', v12), ..., ('Vm', v1m)},
   {('V1', v21), ('V2', v22), ..., ('Vm', v2m)},
    .
    .
    .
   {('V1', vn1), ('V2', vn2), ..., ('Vm', vnm)})
```

The function is not reversible, and neither is the Strong Equivalence Theorem. This is because an arbitrary program of S may require unboundedly large amounts of time and space. Another way of stating the new theorem, reflecting the view of programs as abbreviations of algorithms, is 'Any algorithm can be abbreviated by a program of S.' This is less precise, however, until the particular abbreviation has been defined. It can be defined in terms of abstraction and snapshots to be equivalent to the earlier formulation. Also note that we cannot say that 'Any algorithm is an implementation of the set of computations* of a program of S', because the manner of defining computations of S places severe constraints on the amount of detail allowed; all and only those variables mentioned in the program are included in a computation or snapshot. We can always include more detail in the program than in any given algorithm; it is not so clear that we could fit a program to an algorithm without introducing some extra variables.

Let us turn, then, to the proof of the theorem, which the preceding discussion anticipates. The form to be tackled is

The Strong Equivalence Theorem. Any algorithm A for computing on numbers is an abstraction of a finite subset of the set of computations* of a program P of S.

Note that a finite subset of the computations* of a program of S is an algorithm, so abstraction is defined. The set of all computations* for a program of S is always (countably) infinite, since S's input variables can take on any nonnegative integral value. The finite subset that figures in the proof is the one that has a computation* for each sequence of A, with input values corresponding to the initial states of the sequences of A.

Only a sketch of the proof will be given here because a complete version would require painstaking (and perhaps painful) review of all the i-dottings and t-crossings of the complete formal definition of S. The sketch of the proof is straightforward and convincing enough that the added detail would not add very much. S is a procedural language, so it is helpful to think in terms of one's favourite procedural language or pseudocode.

Given an arbitrary algorithm A, a program P of S can be written which has a separate path for each valid initial state. Once a path is chosen, there are separate subpaths for each valid subsequent state, and so on. At each step in the path, variables are filled in with values reflecting the desired state from A. The structure of the program is like the structure of a forest of trees; each path from a root (initial state) to a leaf (final state) defines one sequence from A. There is one complication to be dealt with, that of

nondeterminism. In the case that there is more than one occurrence of the same initial state or states, a variable not occurring in the abstract state (a new one for each nondeterministic branch) can be used to decide the case. The initial value of this variable should be part of the input, from the perspective of the program P.

As an example, let the algorithm A be

```
{({('X',0),('Y',0)},{('X',1),('Y',1)},{('X',2),('Y',2)}),
({('X',0),('Y',0)},{('X',2),('Y',2)},{('X',3),('Y',3)}),
({('X',0),('Y',0)},{('X',2),('Y',2)},{('X',4),('Y',4)})}.
```

or less formally,

```
X: 0      Y: 0

X: 1      Y: 1

X: 2      Y: 2
_____________________

X: 0      Y: 0

X: 2      Y: 2

X: 3      Y: 3
_____________________

X: 0      Y: 0

X: 2      Y: 2

X: 4      Y: 4
```

The program P can be constructed according to the guidelines given above. A pseudocode version of P

follows which, I claim, could be given an equivalent in S. Of course without knowing the exact S program, we do not know the exact computations or computations*. The idea should be clear, however. More variables would be needed in S (to handle the conditionals, e.g.), and there would be more snapshots in the computations (states in the computations*), reflecting their changes. This is inconsequential as far as the proof goes, since extraneous states and variables can be removed by selection abstractions. Following computations of S, a state is given for the initial state preceding the 'execution' of the first instruction. Thereafter, since the conditionals obscure the line-by-line equivalence between P and a program of S, one state is included for every variable assignment. (Variables are used as in S, with Xi for input and Zi for internal variables, i being a member of N.)

```
IF (X1 = 0 AND X2 = 0) THEN
     Z1 <- 0
     Z2 <- 0
     IF X3 = 0 THEN
          Z1 <- 1
          Z2 <- 1
          Z1 <- 2
          Z2 <- 2
     ELSEIF X3 = 1 THEN
          Z1 <- 2
          Z2 <- 2
          IF X4 = 0 THEN
               Z1 <- 3
               Z2 <- 3
          ELSEIF X4 = 1 THEN
               Z1 <- 4
               Z2 <- 4
```

In this procedure X1 and X2 represent the 'input' in the initial state(s) of A. X3 and X4 are used to decide between the possible second and third states, a decision that cannot be determined from the level of description of A itself. But it is left to the local variables to take on the appropriate values that will be reflected in the abstraction. Z1 and Z2 are given the input values, analagous to the initial state. Z1 will be seen later as X and Z2 as Y in A. In the remainder of the procedure, Z1 and Z2 take on all the possible values of X and Y in A. The sketch of the proof requires abstracting X and Y from the computations* of P.

A finite subset of the relevant computations* of P is:

```
{({('X1', 0), ('X2', 0), ('X3', 0), ('X4', 0), ('Z1', 0),
     ('Z2', 0)},
{('X1', 0), ('X2', 0), ('X3', 0), ('X4', 0), ('Z1', 0),
     ('Z2', 0)},
{('X1', 0), ('X2', 0), ('X3', 0), ('X4', 0), ('Z1', 0),
     ('Z2', 0)},
{('X1', 0), ('X2', 0), ('X3', 0), ('X4', 0), ('Z1', 1),
     ('Z2', 0)},
{('X1', 0), ('X2', 0), ('X3', 0), ('X4', 0), ('Z1', 1),
     ('Z2', 1)},
{('X1', 0), ('X2', 0), ('X3', 0), ('X4', 0), ('Z1', 2),
     ('Z2', 1)},
{('X1', 0), ('X2', 0), ('X3', 0), ('X4', 0), ('Z1', 2),
     ('Z2', 2)}),
({('X1', 0), ('X2', 0), ('X3', 1), ('X4', 0), ('Z1', 0),
     ('Z2', 0)},
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 0), ('Z1', 0),
     ('Z2', 0)},
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 0), ('Z1', 0),
     ('Z2', 0)},
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 0), ('Z1', 2),
     ('Z2', 0)},
```

```
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 0), ('Z1', 2),
     ('Z2', 2)},
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 0), ('Z1', 3),
     ('Z2', 2)},
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 0), ('Z1', 3),
     ('Z2', 3)}),
({('X1', 0), ('X2', 0), ('X3', 1), ('X4', 1), ('Z1', 0),
     ('Z2', 0)},
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 1), ('Z1', 0),
     ('Z2', 0)},
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 1), ('Z1', 2),
     ('Z2', 0)},
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 1), ('Z1', 2),
     ('Z2', 2)},
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 1), ('Z1', 4),
     ('Z2', 2)},
{('X1', 0), ('X2', 0), ('X3', 1), ('X4', 1), ('Z1', 4),
     ('Z2', 4)})}
```

We still must show that A is an abstraction of a finite set of the computations* of P. If we start with the subset of all computations* of P in which the first n Xi are equal to the values for the n labels in some initial state of A (as above), throw out the extra variables not needed by A (using selection of values for each state in each sequence) and change the variable names (change of labels) to match the originals from A, the process is almost complete. We have only to remove the first state of each sequence, along with any interim states giving more details of value changes than in the algorithm (selection of states) to finish the task.

These stages can be illustrated for the example. After applying selection of values, choosing only to show values of 'Z1' and 'Z2', to each state (snapshot) in each

sequence (computation*) of the algorithm (finite set of computations*), we have:

```
{({{('Z1', 0), ('Z2', 0)},
  {('Z1', 0), ('Z2', 0)},
  {('Z1', 0), ('Z2', 0)},
  {('Z1', 1), ('Z2', 0)},
  {('Z1', 1), ('Z2', 1)},
  {('Z1', 2), ('Z2', 1)},
  {('Z1', 2), ('Z2', 2)}),
 ({{('Z1', 0), ('Z2', 0)},
  {('Z1', 0), ('Z2', 0)},
  {('Z1', 0), ('Z2', 0)},
  {('Z1', 2), ('Z2', 0)},
  {('Z1', 2), ('Z2', 2)},
  {('Z1', 3), ('Z2', 2)},
  {('Z1', 3), ('Z2', 3)}),
 ({{('Z1', 0), ('Z2', 0)},
  {('Z1', 0), ('Z2', 0)},
  {('Z1', 0), ('Z2', 0)},
  {('Z1', 2), ('Z2', 0)},
  {('Z1', 2), ('Z2', 2)},
  {('Z1', 4), ('Z2', 2)},
  {('Z1', 4), ('Z2', 4)})}.
```

After applying change of labels, mapping 'Z1' -> 'X' and 'Z2' -> 'Y', we have:

```
{({{('X', 0), ('Y', 0)},
  {('X', 0), ('Y', 0)},
  {('X', 0), ('Y', 0)},
  {('X', 1), ('Y', 0)},
  {('X', 1), ('Y', 1)},
  {('X', 2), ('Y', 1)},
  {('X', 2), ('Y', 2)}),
 ({{('X', 0), ('Y', 0)},
  {('X', 0), ('Y', 0)},
  {('X', 0), ('Y', 0)},
  {('X', 2), ('Y', 0)},
  {('X', 2), ('Y', 2)},
  {('X', 3), ('Y', 2)},
  {('X', 3), ('Y', 3)}),
 ({{('X', 0), ('Y', 0)},
  {('X', 0), ('Y', 0)},
  {('X', 0), ('Y', 0)},
  {('X', 2), ('Y', 0)},
  {('X', 2), ('Y', 2)},
  {('X', 4), ('Y', 2)},
  {('X', 4), ('Y', 4)})}.
```

Finally, application of selection of states to each sequence (omitting the first state of each sequence and the states corresponding to changes in the value associated with 'x') produces the starting point A, completing the sketch of the proof:

```
{({{('X',0),('Y',0)},{('X',1),('Y',1)},{('X',2),('Y',2)}),
 ({{('X',0),('Y',0)},{('X',2),('Y',2)},{('X',3),('Y',3)}),
 ({{('X',0),('Y',0)},{('X',2),('Y',2)},{('X',4),('Y',4)})}
```

To go in the opposite direction, from programs in S to algorithms, is simpler so long as we stay with the one level of description defined by the notion of a computation. The set of all computations for a program P of S is an at most countably infinite set of

computations. Computations are by definition finite, so if P does not halt for some input set, there will be no computation for that set. Each computation can be turned into a computation* by the mapping f given above and, therefore, into an algorithmic sequence. Any finite set of such computations* defines an algorithm. More formally, we have just sketched a proof of:

Theorem. Any finite subset of the set of computations* of a program of S is an algorithm.

CHAPTER VI
SOME IMPLICATIONS

This chapter begins with several observations about the new framework. These include comments on its flexibility and means of constraining it, as well as a summary of the relationship between software and hardware with analogies in the relationship between mental states and physical states. Next, classical and connectionist models are characterised before reconsidering levels in light of connectionism and algorithms. Algorithms are compared to the similar notion of virtual machines as defined by Sloman and, finally, some implications for explanation are explored.

GENERAL OBSERVATIONS

We now have in hand a formal multilevel framework for describing complex systems in a way that is independent of their usual descriptions in terms of particular languages or architectures. The obvious primary implication of this is that it enables us to make strong and direct comparisons between all sorts of systems and states of systems. The systems of interest need not be distinct: the relationships among multiple levels of description of a single system can be elucidated.

States take centre stage. These states are actual states through which the system (at some level of description) passes. They can be contrasted to states of a remote ideal machine whose interim states (between input and output) cannot be aligned with possible states of the actual system. While maintaining the importance of weak input-output (or mathematical) equivalence of system descriptions (or algorithms), the new framework provides a clear notion of strong equivalence. Two systems under such a description are the same when the

possible paths from input to output are the same. These paths, or algorithmic sequences, are equivalent when they contain the same states in the same order. Equivalence in terms of attainable state sequences captures the essence of informal use of 'algorithms' or 'the algorithmic level', especially when these are taken to be the focus of cognitive psychology. This approach is particularly appropriate to cognitive modelling, since the results of experimentally sampling real cognitive systems (i.e. people), or simulations, can be expressed as sequences of states. System states are as indisputable as anything can be as a basis for describing complex systems and incorporating strong equivalence of algorithms into cognitive science (as expressed in Pylyshyn, 1984, e.g., and see Chapter III).

The flexible multilevel framework for strong equivalence can accommodate a surprising number of generalities about a system. For example, it can be shown that the number of states in a sequence is a valid abstraction of that sequence. To prove this, let S be the arbitrary sequence

```
({(l1,v11), (l2,v21), ..., (ln,vn1)},
 {(l1,v12), (l2,v22), ..., (ln,vn2)},
                    .
                    .
                    .
 {(l1,v1m), (l2,v2m), ..., (ln,vnm)}).
```

Combining states one at a time, from top to bottom, gives the valid chain of abstractions of S:

S1 = ({(l1,<v11,v12>), (l2,<v21,v22>), ...,
(ln,<vn1,vn2>)},
{(l1,v13), (l2,v23), ..., (ln,vn3)},
.
.
.
{(l1,v1m), (l2,v2m), ..., (ln,vnm)}).

S2 = ({(l1,<<v11,v12>,v13>), (l2,<<v21,v22,v23>),...,
(ln,<<vn1,vn2>,vn3>)},
{(l1,v14), (l2,v24), ..., (ln,vn4)},
.
.
.
{(l1,v1m), (l2,v2m), ..., (ln,vnm)})

and so on, culminating in the abstraction S' (a one-state sequence) where S' =

({(l1, <<...<<v11,v12>,v13>,...,v1m-1>, v1m>),
(l2,<<...<<v21,v22>, v23>, ..., v2m-1>, v2m>),
.
.
.
(ln, <<...<<vn1, vn2>, vn3>, ..., vnm-1>, vnm>)}).

Selecting the label-value pair for l1, we get another valid abstraction S" =
({(l1, << ... <<v11, v12, v13>, ..., v1m-1>, v1m>)})

Now we can define a function on the value of l1 which maps such nested values to the number of constituent values, giving S''' = ({(l1, m)}), proving the assertion.

Furthermore, if the average number of steps across sequences in an algorithm is known, the function that maps every number to that average can be used to get

another abstraction giving the average number of steps. As an implementation of an algorithm to determine the average, this would be rather unsatisfactory in most contexts (see below on constraining the framework), but it does illustrate the flexibility of the framework.

It does not, however, cover everything. In spite of this, it can be used to facilitate the expression of aspects of system description that lie outside its domain. An important example of this is the constraint on absolute resources from the physical level of concreteness. It may be that a step or a sequence is or must be carried out within predetermined time limits or using a fixed amount of space or storage. Another example in cognitive science is the causal relations between states or parts of states. If program rules are included in state descriptions, for instance, are the rules meant to cause the behaviour underlying later state descriptions? In most cognitive science and computer science applications this is likely to be the case, although pre-compilation software rules could appear in a state description and not have a causal role in the behaviour of a system running the compiled version (unless the evolving system were under consideration -- the original rules have a part to play in causing the compiled rules).

A large part of the motivation for developing the theory of strong equivalence was a foundational one. The framework provides a fundamental definition of algorithms as system descriptions and of the interrelationship between various descriptions. It can be used to clarify comparisons between particular models, between models and that which is being modelled and between classes of models. Putting such comparisons in the framework, even partially, gives a useful perspective and a formal, solid

foundation in place of often muddled speculation about levels of description and explanation.

The overwhelming detail of describing all the states in all the sequences of even simple systems may appear to render this approach useless for more practical applications. To some extent this must be true, and its full practical utility in systems analysis remains to be seen. Translating just a small number of paths at a small number of levels of description for a small number of possible inputs proved surprisingly interesting in the course of developing the thesis. The usual descriptions of classical and connectionist theories often wear their state sequences on their faces. If it is hard to tell what level of description or what sequencing (or lack of it) is meant to be implied by a program or theory, that in itself is of interest. Once theories or portions thereof have been translated into the formal framework, they can be characterised and strongly compared with a rigour previously only possible for (weak) comparisons with respect to a single ideal machine (a Turing machine or a LISP machine, e.g.).

By making states the building blocks from which all sequences (and therefore algorithms) are constructed, processes exist only implicitly as changes between states. This emphasis on states was motivated in part by considering the concept of 'software states' (see Chapter III). Ignoring for the moment the case of specially constructed systems (or evolved systems), higher level descriptions in terms of states and processes are unlikely to map neatly to lower level descriptions in terms of other processes and states, except perhaps using weak mathematical equivalence. For a stronger equivalence, and in particular one which allows the preservation of states through which a system can actually be said to pass, it is problematic to

determine what is a process and what is a state in a manner that will maintain alignment of states across level boundaries. A state containing a '1' and a '0' may be a process at a more detailed level. Core memory, for example, can be described as processes which represent ones or zeroes depending on the direction of current. Yet ones and zeroes are standard low level descriptions of data in computers. At a higher level, programs are often taken to be processes, but they are arguably static, explicit representations.

In contrast, states facilitate a uniform framework for description at every level, with states at one level being expanded to more detailed states or sets of states at lower levels of detail. By using them as the basis of algorithms, the implied processes may be thought of as being expanded into sequences of states at the lower levels, with new interim processes being potentially subdivided further and further, until they are ultimately changes in molecular states, for example.

The framework is intended to be flexible enough to cover common computational language, insofar as it refers to states that are legal abstractions of a detailed algorithm taken from values measured from a physical computer system. In other words, we can say, 'The system is calculating exclusive-or' if it is somewhere between the initial and final states of any algorithmic sequence that has the appropriate initial and final states and that is part of an algorithm containing other sequences which represent states in the calculation of exclusive-or for other inputs. At any particular point in time, if we could freeze the system, we may at best be able to say it is in state i or between state i and state i+1. If the whole system can be viewed in abstraction as an exclusive-or calculator, then it is appropriate to say 'it is calculating exclusive-or' or perhaps even

(depending on the algorithm), 'it is calculating "or" and "and" as it calculates exclusive-or'. In contrast, as was mentioned in Chapter II, it is an empirical question whether a system running a particular program goes through the states implied by that program taken independently. For example the program fragment

```
do i = 1,3
   x = 1
   x = 2
   y = i+x
end
```

may be compiled by an optimising compiler and then run. The optimised system may take on the states of the following sequence (parenthetical comments associate states with lines of the program):

```
i:U  x:1  y:U  (x = 1)

i:U  x:2  y:U  (x = 2)

i:1  x:2  y:U  (do i = 1,3)

i:1  x:2  y:3  (y = i+x)

i:2  x:2  y:3  (do i = 1,3)

i:2  x:2  y:4  (y = i+x)

i:3  x:2  y:4  (do i = 1,3)

i:3  x:2  y:5  (y = i+x)
```

This sequence reflects the common optimisation operation of removing expressions that do not depend on the loop variable (x does not depend on i, in this case) from the

body of the loop. If this is the case, then the algorithmic sequence implied by the above program (by the method of translation described earlier, using statement execution to determine change of states and program variables to determine values in states) is

```
i:1  x:U  y:U  (do i = 1,3)

i:1  x:1  y:U  (x = 1)

i:1  x:2  y:U  (x = 2)

i:1  x:2  y:3  (y = i + x)

i:2  x:2  y:3  (do i = 1,3)

i:2  x:1  y:3  (x = 1)

i:2  x:2  y:3  (x = 2)

i:2  x:2  y:4  (y = i + x)

i:3  x:2  y:4  (do i = 1,3)

i:3  x:1  y:4  (x = 1)

i:3  x:2  y:4  (x = 2)

i:3  x:2  y:5  (y = i + x)
```

For the optimised sequence described above, the statement 'the system is in a loop' would only be accurate after the second step and not for the entire algorithm. Furthermore, the optimised sequence is not a straightforward abstraction of the unoptimised sequence. There is a simple common abstraction:

```
i:1  x:2  y:U

i:1  x:2  y:3  (y = i + x)

i:2  x:2  y:3  (do i = 1,3)

i:2  x:2  y:4  (y = i + x)

i:3  x:2  y:4  (do i = 1,3)

i:3  x:2  y:5  (y = i + x)
```

Here, i and x can be thought of as inputs and y as the output. State changes are associated with changes to either i or y, in keeping with x's role as a constant.

Recalling the search for software states from Chapter III, we can conclude that while software or programs may imply algorithms at various levels, it is an empirical question whether or not any given system -- even one that can be said to run that program -- actually goes through those implied states. If the more detailed description can be seen as an implementation of the less detailed one, the functions defining their relationship may be very complex indeed. It seems misleading to say there are software states or computational states at all; there are simply states of a system (real or hypothetical) described in more or less detail. If a particular set of rules (in a machine language for instance) is stored in the system and used to control the processing, then these rules, along with an indication of the currently active one, will show up explicitly at some levels of description.

Programs or effective procedures are best thought of as abbreviations for algorithms which may or may not be realised in the hardware. Because of the variety of

possible relationships between programs and algorithms it is necessary to be clear about this intended use of a program and also to be clear about the translation that is assumed (e.g., a state is made up of a subset of the variables plus the current program line, or whatever).

Consistent with some common usage (see the discussion of Pylyshyn in Chapter III, for example), a given algorithm can be abbreviated by many different programs. Algorithms may include explicit rules or not (so 'connectionist algorithms' are no longer problematic). They place constraints on the states a system goes through and say something about relative time and space requirements. At the same time, they can be implemented in more detail in an arbitrarily large number of ways and can be realised in an arbitrarily large number of physical devices.

In summary, the relationship between hardware and software is quite tenuous. A program is best viewed as a weakly predictive theory of the output of the machine running the program (or its compiled or interpreted version), a specification of output given input. From the mathematical level perspective, a program defines the relationship between input and output by means of a formal language, implying nothing at all about the interim processing; this is truly the 'what' without the 'how'. At the algorithmic level, a program can be relied on to give the minimal sequential information in addition to the input-output relationship; the input precedes the output on the relative time line. In general it is wrong to assume that the program gives an accurate picture of the states through which the system passes at any more detailed level than that. Assuming the program is actually executable, the hardware or physical machinery realises the input-output relation and minimal ordering (input before output) in absolute space and time

on each occasion that the program is run. Clearly there are many possible algorithmic implementations of the function specified by the program as well as many possible hardware realisations (including a different realisation for each execution). At the same time, there are arbitrarily many algorithmic descriptions of the states through which the machine passes, each of which may be taken as specifying the input-output relationship from the mathematical perspective as well.

If the computational metaphor -- the mind is software to the body's hardware -- is taken seriously, then the implications for cognitive science of the picture just painted are profound. Theories of mental life couched in terms of programs or functional roles of Turing machine states (or states of any other single canonical machine) may be predictive, weak equivalents of cognitive functions, but that is the best they can be. We could not claim that a person was in mental state x (held a belief, felt a pain, etc.) any more than we could claim he or she was in the Turing machine state of reading the tape. The best we could do would be to say that at the end of the day the person behaved as if he or she were in state x, as if he or she held that belief or as if he or she read the infinite tape.

Mental states described by abstract states that are parts of sequences in algorithms have no such restrictions to instrumentalism. We can say that a person was in a mental state x if that state is part of a sequence of states that is a valid abstraction of a much more detailed sequence of states, perhaps in terms of neurophysiological measurements or classical physics. Given the precise abstraction functions for describing the relationship of the two such levels of description, the mental state is an abstraction of one or more lower level states. (One could say 'physical states', but

they are usually already abstractions away from the physical level of concreteness to the algorithmic level.) If both levels are correct descriptions of a physical system, then the higher level mental states are coincident in (relative) time with the lower level states and, in fact, supervene on them. That is, given the functional correspondence between the two descriptions, it can be shown that any change in the higher level description requires a corresponding change in the lower level one.

To prove this, it can be demonstrated that any higher level state supervenes on the one or two states one level down and to which it is related by a single abstraction function. In the first case, the abstract state (call it M1) is a function of a single lower level state M2, because all state abstractions are defined to be functions. So if M1 changed (to M1' which does not equal M1) and M2 remained the same, we would have M1 = f(M2) = M1' for some function f, and this cannot be.

In the second case, M1 is abstracted from two lower level states, M2a and M2b, as part of a sequential abstraction.

Let M2a =
     l1: v12a l2: v22a ... ln: vn2a
and let M2b =
     l1: v12b l2: v22b ... ln: vn2b

By definition of combination of states, M1 =
     l1:<v12a, v12b> l2:<v22a,v22b>... ln:<vn2a, vn2b>

If M1' is also an abstraction of M2a and M2b by combination of states, then it too must be

l1:<v12a, v12b> l2:<v22a, v22b> ... ln:<vn2a, vn2b> which is identical with M1.

To complete the proof, it remains only to note that this sort of supervenience is transitive, so that if a state supervenes on the state(s) from which it was derived one level down, then it supervenes on the states from which it (or they) were derived and so forth, until the lower level in question is reached.

Of course it may be that a change in higher level states results from a change in perspective. This amounts to using different abstraction operations to derive the higher level from lower level states. In this way one value or another, or any combination, can be singled out as the focus of a more abstract state. (This is just one possible abstraction operation -- selection of values; any of the others could be used instead.) In terms of mental states, we might say that a person recalled x, or felt y, or recalled x and felt y at the same time. Each of these descriptions may be different abstractions from different perspectives on the same lower level states and, assuming that this is the case, each mental state description supervenes on the common lower level state descriptions.

The multilevel theory can also be used to describe how two states or sequences of states can be the same at one level of detail and different at another. To borrow an example from Putnam (1988), speakers of English and Thai might both be described as thinking about cats at one level, while at a more detailed level, there are obvious differences in their cat concepts, including how they would say the word for cat in their respective languages. It is true that at extremely detailed levels of description, exactly the same sequence is never carried out twice, by people or by machines. The configuration

of jobs being controlled by the operating system of a mainframe computer may never be the same twice. The locations in memory (virtual or otherwise) may not be either. The same situation is likely to occur with respect to human nervous systems: the configuration of thoughts, concerns, movements and qualia vary constantly, yet at appropriate levels of detail we do think and mean the same things twice. A framework for systematically abstracting away from detail while preserving connections to physical states is essential if anything like functionalism is true or useful (even if 'only' for simulation and explanation and not for instantiation of psychological states). Otherwise we are restricted to talking about only one level at a time with any rigour.

But is the proposed multilevel theory too unconstrained? In particular, consider the generalised rounding and combining-states abstraction operations for states and sequences, respectively, and the subset abstraction operation on algorithms. The abstraction operations for states blur the explicit-implicit representation distinction as well as the related process-data distinction. By combining label-value pairs and using a complex deduction function, a state

x1: A --> B x2: A x3: C

can be abstracted to

y: B

describing (as just one of many possibilities) an inference from A -> B and A (and some other stuff, C) to B. Using duplication of the combined pairs, for example, anything implicit in a state at one level can be made explicit in a more abstract state, subject to the constraints of the functional relationship between such

states. Nothing has changed in the physical system (if there is one), only in the description. I do not see any good reason to restrict such flexibility in general. We can define 'an explicit representation' in the framework as being a single labelled value, automatically tying the idea to a particular level of description.

One argument against placing a priori restrictions on the flexibility of the framework is that quite a lot of flexibility is required just to allow the descriptions normally given to computer systems. These descriptions ignore such complexities in detail as virtual memory and data compression. Graphics applications provide a nice example. A straight line in a graphics system may be represented in a file as two end points perhaps describable by a state

type: 'line' end1: 00342 end2: 00399

At any given time, the line may be displayed on the screen by lighting up a subset of the screen's pixels (which might be described at one level as a matrix of pixels filled in with 1's for 'lit-up' pixels and 0's otherwise). Either of these states may be considered as abstractions or implementations of the other, and why not? Which is more detailed or more explicit depends on the context and the question. A graphic artist pointing to the display and asking, 'How does the machine represent that line?' might consider the file descriptions as providing the implementation, while a programmer asking 'How would the image change if I modified this end point?' might see the particular display as the implementation of a more general programmed description. 'How can I change this endpoint without the line going off the screen?' might require a third perspective, based on points in a design space that does not necessarily match up with the display screen.

(The design space could be three-dimensional while the screen shows only two dimensions.)

On the other hand, if there were theoretical reasons in a particular context for limiting the power of abstraction operations, such restrictions are more likely to be formulable in the new formal framework than in the usual vague language of levels and implementation. In cognitive science, it makes sense to disallow functions that require information about all possible outcomes of an algorithm. This would prevent viewing some systems as driven by large tables. Related further restrictions could be placed on any constant used in abstraction functions, forcing higher level values to depend more directly on lower level ones. This would avoid the result that any algorithm has an abstraction that includes a fixed set of rules (see below).

Constraints on algorithms also arise in context from the physical and mathematical levels of concreteness. The mathematical level does not add much to a coarse input-output algorithm, though it may abstract away from limits of finiteness or otherwise put the input-output relationship in different terms. The physical level constraints on resources are another matter. See Rosen (1985) for an example of a more detailed foundation for the direct abstraction of an algorithmic level description (or model in his terms) from a physical system (or natural model). According to him, an abstraction or model is the result of observation or measurement of a physical system. By grouping the measured values by simultaneity in time and by meter across time, something like an algorithmic sequence is obtained. So we can say that a new state in a sequence might be motivated by a change in value or by the passage of a given amount of time regardless of measurable value change. The only constraint on the meter is that it

produce a function of any more basic, individual, measurements it takes, as in the abstraction operation of generalised rounding; further abstractions can be made by applying functions uniformly across sequences. Similarly, states can be combined by a meter, calculating a difference between values at times ti and ti+1, for example. In these cases, a more detailed algorithmic description of the system is possible in terms of the more basic values measured. The amount of detail is limited by the technology of the meters, e.g., how quickly they can act or to what degree values can be differentiated.

As noted earlier, absolute time and space restrictions may be added to an algorithmic description. If each step requires at least n seconds and the whole sequence takes up at most 10n seconds, then a sequence of more than ten steps is not the right description in the context.

In order to focus on such thorny problems as the relationship between software and hardware, implicit and explicit representations and levels of description of algorithms, learning and development issues have been ignored. Where do they fit into the picture? While not singled out as such, a lot of learning (especially machine learning, including automatic programming and connectionist learning algorithms) is covered automatically by the framework. A connectionist system, such as one of the exclusive-or examples from Chapter III, is likely to be trained (or programmed) and run on a conventional von Neumann system. The training process can also be described in terms of algorithms at various levels of description, with changing weights at some levels or with changing exclusive-or calculations (becoming more and more accurate as time passes).

Within the new framework, learning and development are just state changes like everything else. There is no mystery about reflection or changing the process instead of the data. Of course, there are valid questions about how the learning takes place at a more detailed level or if it can be realised under appropriate time and space restrictions, but again, this is no different to questions regarding any kind of state changes over time. A system that evolves over time can be treated as a longer, more complex algorithm. Another possibility is to treat systems attained at various stages in the development process as separate, comparing them like any two distinct systems.

It should be noted, however, that the framework as it stands requires that all states and sequences in an algorithm be defined over the same set of labels. In some cases a description of a developing system could use a combination of all the appropriate labels from each stage of development, and perhaps this is no worse than the cumbersome detail of the whole framework. However, this may not produce a valid abstraction of an evolved or evolving physical system. As in systems not characterised by learning, the framework says nothing about causation. Important causal theories about how a system got to be the way it is (such as evolution-based theories or developmental ones) can be stated with respect to descriptions in the framework, but they lie outside its domain by definition.

CHARACTERISATION OF CLASSICAL AND CONNECTIONIST THEORIES

Theories categorised as either classical or connectionist fall into the algorithmic level of concreteness. That is to say, they generally make more than just behavioural or input-output claims; they place constraints on the relative time sequence of steps, for example. While

they may be viewed as defining a mathematical relationship between input and output, they place restrictions on theories that are lower level in terms of detail and also on those higher level theories (again in terms of detail) that may be valid abstractions of them. In addition, they may be implemented in arbitrarily many ways, by spelling out the details differently, and they may be realised in physical hardware in arbitrarily many ways, so long as the lower level detailed descriptions of such implementations or realisations are valid algorithmic implementations.

Using state-based algorithmic descriptions, some general distinctions do emerge between classical and connectionist theories, but it should be emphasised that these are generalisations. There are no hard and fast rules for defining a distinct equivalence class of connectionist or classical theories or systems. With that caveat in mind, the differences are as follows. Connectionist algorithms, at their usual levels of description (in terms of state changes over activation vectors typically) tend to be more nondeterministic and to have changes to more values in each transition from state to state. The latter characteristic may be shortened to more distributed changes. Classical algorithms, on the other hand, tend to be more deterministic with less distributed changes, at their usual levels of description.

The underlined phrases bear further examination. 'At their usual levels of description' captures the largely valid stereotype of the distinction between connectionist algorithms as theories in terms of numeric activation values associated with nodes that stand for features or subconcepts (Smolensky, 1988), as opposed to classical theories in terms of, say, inferences from a set of natural-language-like beliefs. 'Determinism' is the

ordinary notion whereby a given state completely determines the next state. In a deterministic system, there is only one path from a particular initial state to a final state. In a connectionist system, it is more likely that there will be many paths from a particular input state to a particular output state. (In classical system descriptions, there tend to be fewer paths.) In general, the next state cannot be predicted from a given state. Nondeterminism is of course closely related to parallelism. The important point to notice is that a set of algorithmic sequences can be seen to be nondeterministic, or to have distributed value changes, whereas it may or may not be physically realised in hardware usually described as parallel. It is a mistake to refer to an algorithm as sequential or parallel.

'Distributed change' is closely related to distributed representations in the usual connectionist phraseology, whereby a change to a single value (a concept, e.g.) at one level is seen to be a change to many values (subconcepts, e.g.) at another. In contrast, classical systems tend to describe fewer changes at a time. If more than one value does change in a single state transition in a classical description, it is likely to happen on a much smaller scale than in a connectionist system. That is, a smaller number of simultaneous changes will occur a smaller number of times.

It is interesting to note that after stripping away concerns with causation and semantics and then applying the uniform framework of state transitions, we arrive at a characterisation of connectionist systems that is a simple shift of perspective from the usual one. Rather than parallel distributed processing (PDP) systems, they are seen to be nondeterministic distributed-change (NDC?) algorithms. This shift of perspective clarifies the nature of implementation. While one might expect a

parallel system to be realised in parallel hardware, a nondeterministic algorithm may turn out to be implemented in a deterministic algorithm at a lower level of detail. This would happen, for example, when values distinguishing all input-output paths at the lower level were just those abstracted away from at the higher, nondeterministic level. Similarly, we normally think of distributed representations as being inherently lower level than local representations, but using the duplication abstraction operator, it is possible to have many views (subviews?) of a value at a higher level. This is what a multilevel description of a data compression system is likely to involve. At the same time, since we do appear to have parallel brainware constraining materialist theories of psychology from below, it makes sense to have a nondeterministic level of description (at least one) between the level of molecular states and any possible deterministic symbol system level.

How do differences in nondeterminism and distribution of change relate to the more common characterisation of connectionist systems in terms of flexible generalisation and absence of explicit rules, among other things? Neither sort of classification provides hard and fast rules, and both trade on the increased complexity of analysis required for typical connectionist systems, as well as the fact that these algorithms tend to be viewed from many levels at once -- including activation state changes along with a more classical level and the input-output-only level (either mathematical or a coarse level of algorithmic detail).

A recent summary of the appealing features of connectionist systems from a philosophical vantage point is given by Clark (1989, p. 184). The list is presented as a set of constraints on systems of philosophical and

psychological interest which PDP systems can satisfy, rather than as any strict division between connectionist and classical theories. I repeat it here, regrouped and with my comments interspersed.

> A powerful learning capacity,

> Continuity with the kind of architectures dictated in evolutionarily basic cases (i.e., the constraints imposed by the gradualistic holism of evolutionary change)

These two items lie outside the scope of this thesis, strictly speaking. Developmental issues of both these kinds (individual learning and evolution of the species) are fascinating and important. They can also confuse the explication of levels of description of a particular system at a particular stage in its development.

> A capacity to deal with unanticipated situations (e.g., to generalize along unexpected dimensions)

Any computer model is designed to handle certain inputs in a specified way. Other inputs, if they are allowed at all, can be said to give rise to 'unanticipated situations'. What happens in these cases is determined by a more detailed implementation of the algorithm. The more detailed algorithm that shows what happens at the appropriate level of interest may be considered to be of theoretical interest or it may be irrelevant 'mere' implementation. It is customary to view connectionist systems as having more (and more detailed) levels of theoretical interest. They are often viewed as implementations of classical theories with debate arising over whether the implementation is of theoretical interest or not.

There is more to say about the idea of 'unanticipated situations' and flexibility in general. One advantage of

viewing algorithms as sets of sequences of states at various levels is that irrelevant implementation details can be omitted altogether. We can define an exclusive-or algorithm as follows (sequences are separated by horizontal lines):

```
x : 0     y : 0     'x xor y' : U
x : 0     y : 0     'x xor y' : 0
-------------------------------------
x : 0     y : 1     'x xor y' : U
x : 0     y : 1     'x xor y' : 1
-------------------------------------
x : 1     y : 0     'x xor y' : U
x : 1     y : 0     'x xor y' : 1
-------------------------------------
x : 1     y : 1     'x xor y' : U
x : 1     y : 1     'x xor y' : 0
```

Here there are no surprises. If x and y are input as 0.5, the algorithm has nothing to say about it. Taken as a theory of the behaviour of a system, the algorithm makes no prediction. If it is possible somehow to input 0.5's to the system, then we can say that the outcome is dependent on the implementation, or a valid and appropriate description at a lower level. It is somewhat misleading to say that the system may 'generalize along unexpected dimensions' or, as is common in computer manuals, that the results are unpredictable. At the levels of detail suitable to answer the question 'What happens if you input 0.5's?' in this context, the result is predictable and the 'generalisation' expected. Again, if that level is not part of the theory then it is irrelevant, possibly misleading, mere implementation. If however it is part of the theory, then the detailed algorithm should be included in the theory and the situation is no longer totally unanticipated.

There is nothing magical about such details that are specified at one level as opposed to another. They are built into the system and there is no guarantee they will be natural, desirable additions to the theory or something rather less welcome. If the result of entering 0.5's to the exclusive-or system is an output value of 0 or 0.5, we could argue that these are sensible generalisations, but this would be dependent on the given context. The system may be incapable of producing any output outside the 'valid' range of 0's or 1's. In some cases it may be better to have some indication of 'invalid input' to indicate that the input has gone beyond the level at which the theory is intended.

Because connectionist theories tend to be viewed at several levels and are set up to be seen in that way, and because their input and output ranges tend to be very restrictive, many detailed variations or generalisations seem sensible. These may be incorporated into a multilevel theory even if they were not there to begin with. The fact that such systems are frequently described in terms of units and connection weights, along with the complexity caused by their nondeterminacy and distributed change at that level, makes them difficult to analyse at a glance. This no doubt adds to the impression that their outputs are sensible generalisations along unexpected dimensions even though the theory predicted them all along.

> General flexibility in the use and recovery of stored data,

> The capacity to shade meanings according to context, to create schemata on demand, and so on

As above, apparent flexibility can result from the possible input and output modes and from the multiple levels of theoretically interesting descriptions. Some

examples drawn from database-like PDP systems are also typical examples of classical relational database capacities. Retrieval of an item by its attributes, or content addressability, is one commonly cited feature of both. Some generalisations built into connectionist systems, such as probabilistic attribute values, are not usually built into relational data models (though they could be). At the same time, generalisations available in most relational database systems include counting ('How many x's have attribute y = "a"?') and more complex logical combinations of attributes ('List all x's with attribute y = "a" or attribute y = "b"'), as well as simple mechanisms to add more attributes or more individuals. As database and schemata systems, connectionist versions can be viewed at (at least) two levels, with one level approximating a traditional system (which is likely to be a valid abstraction in the new framework) and another being a finer-grained, more nondeterministic algorithm with distributed change. At the lower level of activation values, there is no surprise or subtlety other than that caused by the complexity of the system. It is only from the relative differences in the two levels that we get shades of meaning.

None of this is meant to belittle the stereotyped higher or lower classical or connectionist perspectives. What this comparison does show is that all levels are potentially of interest. Beyond that, it shows that a further advantage of a multilevel approach is that it provides greater opportunity for comparisons between levels.

> Rule-describable behavior without explicit fixed rules

While any algorithm can be given an implementation which incorporates a fixed set of rules (ignoring for the moment any bottom-up constraints) by appending them to every state, the inverse is also true. That is, every algorithm or algorithmic sequence has a valid abstraction that includes a fixed set of rules.

To prove this, we take the fixed set of rules to be a constant c and show that S1 =

x: v1 rules: c

x: v2 rules: c

is a valid abstraction of S2 =

x: v1

x: v2

S2 is meant to stand in for an arbitrary sequence; any other would do as well. The only way an abstraction can have more values in each state than its implementation is through the duplication abstraction (or a chain of abstractions including duplication). Since S2 has only one value in each state, we duplicate that value. (In an arbitrary sequence, pick any value; the set of labels is finite so this is no problem.) After duplication, we have an interim algorithmic sequence S2' =

x: v1 x2: v1

x: v2 x2: v2

which is an abstraction of S2, and we must show that S1 is an abstraction of S2'. We combine two steps at once, taking the label-changing abstraction to change 'x2' to

'rules' and finding a function that maps v1 to c and that also maps v2 to c. Since there is no difficulty in finding such a constant function, S1 is a valid abstraction of S2' and therefore of S2.

Of course it is this sort of contrived mapping with a large constant element that we may want to rule out in constraining the flexibility of the multilevel framework. At the same time, we again blur the distinction between connectionist and classical systems; the above argument applies to either. The important point, once again, is the need for clarity about the level at which a theory is stated and about what it means to be an explicit representation at a level.

> Robustness (a tolerance of local hardware damage),
> Fast sensory processing,
> Economy of storage and retrieval

While it is true that many realisations of connectionist systems exhibit these traits, there is a danger of confusing algorithmic and physical levels of concreteness, especially when citing such features as reasons to avoid classical theories. Due to the relative and qualitative nature of the differences between classical and connectionist algorithms, Fodor and Pylyshyn (1988, p.54) are right to point out that the standard connectionist objections to classical architecture

> depend on properties that are not in fact intrinsic to Classical architectures, since there can be perfectly natural Classical models that don't exhibit the objectionable features. (We believe this to be true, for example, of the arguments that Classical rules are explicit and Classical operations are 'all or none'.) ... The objections are true of Classical architectures insofar as they are implemented on current computers, but need not be true of such architectures when differently (e.g.,

> neurally) implemented. They are, in other words, directed at the implementation level rather than the cognitive level, as these were distinguished in our earlier discussion. (We believe that this is true, for example, of the arguments about speed, resistance to damage and noise, and the passivity of memory.)

Fodor and Pylyshyn's notion of cognitive level is the problematic middle level similar to Marr's algorithm and representational level and Newell's symbol level (see Chapter II). They claim to be stating their theory at the one right level of detail and relegating connectionist theories to mere implementation (which is also used ambiguously between lower levels in terms of detail and in terms of concreteness). The connectionists (notably Rumelhart and McClelland -- once again, refer to Chapter II) also claim the algorithmic high ground, pushing Classical theories to an idealised higher level closely akin to Marr's computational level. Clark too sees folk psychology as descriptive (I would say predictive), defining input-output relationships without making any commitment that interim states are abstractions rooted in lower level states measurable from a physical system.

The answer to this dilemma is clearly stateable in the new framework: both classical and connectionist theories are algorithmic in the new sense. It is an open question for any two given theories whether one can be seen as a valid abstraction (with respect to detail) of the other. Both sorts of theories make claims at higher levels of detail most likely, as when connectionist theories claim emergent schemata. (In fact emergent structures can be defined as representations or values in an algorithm that is a valid abstraction of a lower level algorithm.) Any two given algorithms make claims at the mathematical level of input-output profiles as well. Theories of both types potentially make claims that they

are valid abstractions of human systems. While the gaps in understanding of neural algorithms remain large, more and more (typically connectionist) theorists are concerned with valid abstractions tied to physical systems. (For example, see Eichenbaum and Buckingham's, 1990, hippocampus modelling and Beer's recent cockroach simulation reported by Sterling, Beer and Chiel, 1990.) That all these theories are algorithmic is true insofar as they make claims about interim states of abstract systems, including the relative time-course of the states. Fodor and Pylyshyn claim that something like the sequence

beliefs: A and B conclusion: U

beliefs: A and B conclusion: A

lies behind some inferences, while connectionist claims tend to be in terms of variation of activation values and higher-level groupings thereof. The new formal framework provides a vehicle for clarifying just what these claims amount to.

Reference to absolute time and space attributes, however, is an easy indication that we are out of the realm of algorithms. When algorithms or theories or architectures are compared in terms of such absolutes (speed, space or efficiency, for example), the red flags should go up. This happens frequently. Clark, e.g., says (1989, p. 88) of the Jets and Sharks (a toy connectionist database-like system), 'Storing the information in a network of the kind just described is a very natural, fast, and relatively cheap way of achieving the same result.' (my emphasis). But as Fodor and Pylyshyn (pp. 55-56) comment,

> ... absolute constraints on the number of serial steps that a mental process can require, or on the time that can be required to execute

them, provide weak arguments against Classical architecture because Classical architecture in no way excludes parallel execution of multiple symbolic processes. Indeed, it seems extremely likely that many Classical symbolic processes are going on in parallel in cognition, and that these processes interact with one another (e.g., they may be involved in some sort of symbolic constraint propagation). Operating on symbols can even involve 'massively parallel' organizations; that might indeed imply new architectures, but they are all Classical in our sense, since they all share the Classical conception of computation as symbol-processing.

Similar arguments apply to claims that connectionist systems have their virtues in being hard-wired or physically distributed. Physical constraints can be added onto an algorithmic description, as predictions from a top-down theory or descriptions drawn from bottom-up measurements. Fodor and Pylyshyn (p.55) assert a version of the latter in the following:

> ... the fact that individual neurons require tens of milliseconds to fire can have no bearing on the predicted speed at which an algorithm will run unless there is at least a partial, independently motivated, theory of how the operations of the functional architecture are implemented in neurons. (their emphasis).

The recent upsurge of neurally-based theories of locomotion, memory and so on provides a much needed bottom-up component to cognitive research. Good theorising, like good design, requires the matching of constraints from the top and from the bottom, as advocated by P.S. Churchland (1986) as 'co-evolution of theories'.

In summary, emphasising the multilevel approach against any uniform single-level one (as advocated here and by Clark, 1989, as well as by Churchland) allows an ecumenical view of the classical versus connectionist

debate. The state-based version in particular allows comparisons across levels and architectures, resulting in a focus on the many similarities between classes of complex systems and the difficulty of nailing down any general differences. Given the weak constraints on the abstraction-implementation relationship, it may well turn out that the classical, symbolic approach to cognition is a valid abstraction of neuronal level algorithmic descriptions. (Even positing fixed rules as a valid abstraction adds nothing.) In the flexible multilevel framework, it is not a very strong claim, though it is an interesting and empirical one.

Smolensky (1988, p. 11) uses the terminology of Haugeland (1978) to distinguish the classical symbolic paradigm and the connectionist subsymbolic one:

> A symbolic model is a system of interacting processes, all with the same conceptual-level semantics as the task behavior being explained ... this systematic explanation relies on a systematic reduction of the behavior that involves no shift of semantic domain or dimension.

But in fact the latter claim is true only of a subset of systems called 'systematic hierarchies'; these include the 'information processing systems' that Haugeland equates with cognitivism (see Chapter II for a discussion of Haugeland's levels and terminology). In arguing that connectionist models are not systems and therefore provide explanations and descriptions in terms of 'good old-fashioned numerical vectors' (p. 11), Smolensky ignores a third possibility. That is that connectionist models are suitably described as non-hierarchical systems. Haugeland says of systems in general (1978, p. 247 - 8) that

> ... objects with abilities that get systematically explained must be composed of

> distinct parts, because specifying interactions is crucial to the explanation, and interactions require distinct interactors. Let a system be any object that is explained systematically, and the functional components be the distinct parts whose interactions are cited in the explanation. In a system, the specified structure is essentially the arrangement of functional components such that they will interact as specified; and the specified abilities of the components are almost entirely the abilities to so interact, in the environment created by their neighboring components. Note that what counts as a system, and as its functional components, is relative to what explanation is being offered. Other examples of systems (relative to the obvious explanations) are radios, common mousetraps, and (disregarding some messiness) many portions of complex organisms.

The distinction between the two sorts of systems is clear in the following excerpt; again, note the similarity of connectionist explanations to the non-hierarchic variety of systematic explanation (p. 250 - 251):

> In explaining a system, almost all the abilities presupposed are abilities of individual components to interact with certain neighboring components in specified ways. Since intricate, interdependent organization is the hallmark of systems, the abilities demanded of individual components are often enough themselves rather sophisticated and specialized. Conversely, since systems typically have abilities strikingly different from those of any of their separate components, systematic organization is a common source of sophisticated and specialized abilities. These considerations together suggest that very elaborate systems could be expected to have smaller systems as functional components. And frequently they do - sometimes with numerous levels of systems within systems. For example, the distributor system of a car is a component in the (larger) ignition system, which, in turn, is a component in the complete engine system.

He goes on to mention a related discussion of hierarchies in Simon (1969). In Simon's terminology, a connectionist system might be viewed as a 'flat' hierarchy, with the main system decomposing into many interacting modules at the next level, modules which do not decompose much further. In contrast to Smolensky's claim, explanations given by connectionist models do fit into Haugeland's category of systematic explanations. Such explanations in terms of the functional components of a system might be put into the framework of state-based algorithms (possibly abbreviated by parallel programs) if such descriptions were reflected in the physical states of the underlying system.

LEVELS REVISITED

It is worth revisiting levels, since their detailed treatment in Chapter II preceded the introduction of the new definition of algorithms and specific consideration of connectionism. Clark's summary of the 'received attitude' towards 'levels of description of connectionist systems', based largely on Smolensky (1988), is a good point of departure. Unlike the standardised multilevel framework in terms of states, labels and values, here each level has 'its own characteristic vocabulary' (Clark, 1989, p. 188).

Level 1, the numerical level, fits in easily with algorithms as sets of state sequences.

> ... the theorist can give a precise characterization of the state of such a system at a particular time by stating a vector of numerical values. Each element in the vector will correspond to the activation value of a single unit (p. 188).

(Of course the new framework allows more detail than this: interim states a node passes through to compute

its activation function may be expanded at a lower level, for example). States may be summarised or related by equations over activations, or weights if these are included.

Level 2, the subsymbolic level, can be seen as a change-of-labels abstraction from level 1, with activation values given new labels according to a (sub)semantic theory at this level. Without the semantic theory, it is just a renaming of the numeric level, with its characteristic (but not required) nondeterminism and distributed change.

Level 3 is cluster analysis, or 'the partitioning trees created by performing a cluster analysis on a network.' This is Clark's addition to the hierarchy, inspired by an interesting analysis of NETtalk by Rosenberg and Sejnowski (Rosenberg, 1987; Sejnowski and Rosenberg, 1987).

NETtalk is a connectionist network that was trained to associate sounds (phoneme encodings) with input letters (given in the context of surrounding letters from English words). There is one input vector (or layer) of units which are not connected to each other but 'feed forward', each sending output to each of the units at the next intermediate (hidden) layer. The hidden layer has units which again are not interconnected to each other but feed forward to the output units which ultimately contain the phoneme encodings.

Cluster analysis is actually a set of statistical techniques for finding equivalence classes, or clusters, among the input data (Lorr, 1983). Rather than producing a single level view, each method potentially suggests a number of different abstractions of the input and output relationships. In the work described in 'Parallel

Networks that Learn to Pronounce English Text' (Sejnowski and Rosenberg, 1987) an agglomerative method using complete linkage was used, while in another analysis (Rosenberg, 1987) similar results were obtained using a slightly different (centroid) bottom-up method. For each letter-to-phoneme correspondence, the activation values of the hidden units were averaged and these average vectors were 'clustered'. In the former (p. 157) it was reported that

> the most important distinction was the complete separation of consonants and vowels. However, within these two groups the clustering had a different pattern. For the vowels, the next most important variable was the letter, whereas consonants were clustered according to a mixed strategy that was based more on the similarity of their sounds.

This suggests some possible abstractions: most obviously the system can be seen as a two-step vowel/consonant classifier, taking letters as input and mapping them to a vowel category or a consonant one. A different perspective that may or may not be a valid abstraction of the system sees letters classified as either vowels or consonant-sound clusters, with more detail in the latter category than just a general consonant class. A detailed view including a middle step and the middle layer might show how different units contribute to different analyses (perhaps using the abstraction operator of duplication to separate out different views of the same unit). Such an analysis may be possible even without statistical analysis. As Sejnowski and Rosenberg state (again on p. 157),

> It was apparent, even without using statistical techniques, that many hidden units were highly activated only for certain letters, or sounds, or letter-to-sound correspondences. A few of the hidden units could be assigned unequivocal characterizations, such as one unit that

> responded only to vowels, but most of the units participated in more than one regularity.

Cluster analysis provides no guarantee that the abstractions indicated by one or more of its techniques will be valid abstractions of the more detailed description at the level of activation values. In the particular approach discussed by Clark, the various groups of hidden vector values were averaged before being subjected to analysis at all, so it may be that the categorised values are never actually taken on by the system in the strong sense required by the algorithmic framework.

Level 4 is the symbolic AI level. This is meant to be the classical, artificial intelligence side of Smolensky's conceptual level (which also includes natural language terms and folk psychology). Clark summarises this level as involving 'any construct of classical AI, e.g., a schema, production, prototype, and so on.' (Clark 1989, p. 194). Once again, in the new framework this is likely to comprise many levels or to be one of many possible levels, depending on the context and question at hand. It may include the programs or productions or schemata themselves in the description, or just the results of their application. If such descriptions are valid abstractions of a lower level, then they are just as correct as any lower level, only less detailed. In this way, even the so-called 'precise' numeric level may be correct but less detailed than a description giving the steps toward calculating an activation function or details of implementation in a von Neumann system.

The folk-psychological level, or level 5, is the other half of Smolensky's conceptual level, and it is an open question whether or not this is a valid abstraction of humans or models of them in particular contexts. Clark

describes (but does not adhere to) a view that constrains a valid abstraction to include only the case in which 'words used in a belief ascription' do have

> discrete, recurrent analogues in the actual processing of the system. Thus the word 'chair' will not have a discrete analogue, since 'chair' will be represented as an activation vector across a set of units that stand for subsymbolic microfeatures, and it will not have a single recurrent analogue (not even as an activation vector), since the units that participate and the degree to which they participate will vary from context to context (p. 196).

The new multilevel framework advocated here is less constrained, allowing for abstractions (including duplication and combination of values) of such 'pure distributed representations' to define sameness of belief at some levels of description in some contexts.

ALGORITHMS AND VIRTUAL MACHINES

Since Clark is a leading advocate of 'multiplicity of mind', a promising multilevel approach to cognitive science, an obvious question to ask is exactly how he spells out a theory of levels and their interrelationships. In Microcognition (1989), he leans toward an explication in terms of 'virtual machines': On pages 128 - 129, he considers

> a possible multiplicity of virtual cognitive architectures. The idea is that for some aspects of some reasoning tasks, we might be forced to emulate a quite different kind of computing machine... The argument I develop will urge that we resist the uniformity assumption in all its guises. Instead, I endorse a model of mind that consists of a multitude of possibly virtual computational architectures adapted to various task demands. Each task requires psychological models involving distinctive sets of computationally basic operations.

If we replace 'virtual machines' and 'architectures' with levels or algorithms, this appears quite similar to the massively multilevel formal framework. While both approaches arise out of a reaction against uniform, single-level cognitivism and both draw on computational theory, there are some differences. For one thing, the state-based framework has been developed (shown to be formalisable and applicable) to a greater degree. To the best of my knowledge the situation for virtual machines has not progressed much beyond 1984 when Pylyshyn (1984, p. 91) noted of them that '... the idea has not been formally developed to a high degree in the study of semantics of programs...' I suspect one reason for this is the complex, often misunderstood relationship between software and hardware, along with the fact that notions of virtual machines are bound up with limiting ideas of programs, emulation and classical architecture. Mappings between virtual machines, like current definitions of theory reduction, require mappings between processes as well as between states. This tends to be mediated by a compiler or interpreter, ending up with at best a weak equivalence between like functions described as running on different levels of the architecture.

Pylyshyn defines a virtual machine quite clearly in this vein, speaking of (1984, p. 91)

> defining an abstract virtual machine whose state transitions correspond, in well-defined ways, to the significant features of changes produced by commands, procedures and other expressions in a program's text.

He equates the virtual machine to the available operators (p. 115) or to a computer's programming language (p. 260). This is echoed more recently by Clark (1989, p. 131):

> The pervasive idea in computer science of a virtual machine is precisely the idea that a machine can be programmed to behave as if it were operating a different kind of hardware.

and (p. 12)

> A virtual machine is a 'machine' that owes its existence to a program that runs (perhaps with other intervening stages) on a real, physical machine and causes it to imitate the usually more complex[1] machine to which we address our instructions.

The way in which Clark applies the notion of virtual machines, however, turns out to be much more like the perspectival view advocated in this thesis. For example (p. 138),

> In the special kind of case in which actual discrete environmental structures (or mental models thereof) are manipulated according to explicitly formulated rules or heuristics, we have a virtual machine that recapitulates the processing steps of a conventional model. That is, the kinds of operations we perform on real, external symbolic structures (and hence the kinds we use in any mental model of the same) are just the operations found in a conventional processor, e.g., completely copying a symbol from one location to another, deleting, adding whole symbols (e.g., 'cup' to a list), and matching whole symbols. In these special cases, therefore, the conventional model is not any kind of approximation to the truth; it is the truth.

The importance of aligning states that the system (the person and various external props in this case) goes through at various levels of abstraction show up in this example. Clearly Clark is after something stronger than

---

1 I would say 'simpler' rather than 'complex', at least in computer science, where virtual machines allow viewing memory as one contiguous chunk, e.g.

the weak equivalence guaranteed by 'a program that runs (perhaps with other intervening stages)'. The conventional operations he is talking about appear to be transitions between actual states through which the system passes, viewed at an appropriate level of abstraction in terms of detail. Similarly, he is taking an equally high level and abstract view of the conventional system in terms of copying symbols and so on.

If we go to Sloman (1984a)[2] we find a more detailed explanation of virtual machines. Sloman states (1984a, p. 10),

> A virtual machine is a structure which can undergo various changes of state, where the state is defined relative to a certain class of descriptions, and certain transitions from one state to another are defined as legal.

This differs from the definition of an algorithm defined over a set of labels just in that the emphasis is on processes (transitions are separated out and defined as legal, independently of their occurrence in sequences) and infinity or unboundedness may be allowed, along with weak equivalence due to programming.

Describing the relationship between virtual machines, Sloman states (p. 10 - 11),

> one virtual machine can be implemented (or embodied) in another virtual machine which satisfies far more detailed state descriptions. The same virtual machine may be embodied in different physical machines. Different virtual machines can be embodied in the same physical machine.

---

2 Clark refers to Sloman in his first mention of virtual machines, but it is to a less detailed paper (1984b).

Here we could substitute 'algorithm' for 'virtual machine' and see the similarity between the two notions, but implementation is also defined in terms of general states and transitions (out of context) and can also involve programming. On p. 11 we have

> Virtual machine A is implemented in B by specifying a mapping M, where M maps descriptions of the states of B into states of A, and maps 'legal' transitions of B into primitive legal transitions of A. Thus A may only refer to integers as possible values of variables, whereas B may have far more memory locations each capable of having a binary state. A collection of bits of B may be taken to represent a single integer of A. It is not necessary that every state of B correspond to a state of A. For instance, the process of giving a variable of A an integer value may involve changing several locations in B, in sequence, and some of the intermediate states of B need not represent any state defined in A. This also illustrates the fact that a primitive (unanalysable) transition of A may be represented by a succession of state transitions of B.

By focussing on explicit sequences of states or abbreviations of such sequences, the new framework in terms of algorithms allows complete formalisation of the relationship between two levels in a much simpler, more straightforward way.

Sloman also includes some programming (that I would consider part of the software development process) as part of implementation, though he is careful to separate it in the definition. On p. 11, he states

> Machine B implements A relative to a mapping M. Relative to a different mapping, B may implement a different machine. Usually implementation is done by taking a fairly general machine B, then producing a more specific machine B1 by programming it ('specification'). B1 implements A1 relative

> to mapping M1. A different specification of B may produce a machine B2 which implements another machine A2 via a different mapping M2.

The term 'implementation' may refer to the combination of program or specification and mapping, as in

> Moreover, there may be different ways of consistently implementing A in terms of B, via different programs and mappings. (p. 11).

Sloman's definition is intended (and I believe succeeds) as a philosophically respectable definition of virtual machines as they are used in conventional practical computer science. Inherent in this approach is a relatively small number of virtual levels that can be defined independently of context, such as (1) the virtual machine defined by an individual LISP program running on (2) a virtual machine defined by a LISP interpreter running on (3) an IBM-assembler virtual machine, for example. This comes through in Sloman's writing as in (p. 12),

> There may be several levels of implementation. An interesting question is to what extent and in what way either learning processes within an individual or evolutionary processes can bring it about that the number of layers of implementation increases.

This stands in contrast to the massively multilevel theory of algorithms wherein a new level can be 'created' by a shift in perspective and, for any given system, there is no pre-determined number of levels of interest. A new question in a new context may bring a previously unnoticed level into the limelight.

The influence of conventional, classical virtual machines is seen further in Sloman's comment on page 16:

> The study of different sorts of parallelism and their properties is now in its infancy ... The implications of this sort of distinction are at present hardly understood, though it seems clear that at least in the more complex animals, brains are of the massively parallel type.

In fact, the adjustments needed to adapt Sloman's definition of virtual machines and implementation to cover parallel systems are arguably just the modifications of the new algorithmic framework. Separating out processing and programming issues leaves mappings between states (and between transitions only implicitly) in context, which are applicable to any sort of system at any algorithmic level. The limitations on number of levels in conventional virtual machine theories arise in Sloman's discussion but do not logically follow from his definition of implementation in terms of mappings.

Perhaps due to the simpler, stronger equivalence of states at various levels underwritten by my theory, I disagree with Sloman's position in this paper with respect to the role of implementation. He maintains that

> Questions like whether the abstract machine A implemented on the physical machine B, possibly via many intermediate layers, really IS B, or whether the state Sa of A which exists at time t really IS the state Sb of B which exists at time t are non-questions. The ordinary conception of identity was not designed to cope with such complex relationships as this: the ordinary meaning of 'is' provides no criteria for settling the issue. Moreover, nothing hangs on how it is settled. If we know that the relation is one of implementation what more is there to know? (p. 12 - 13).

Aligning states that a system goes through at different levels of description is important, however, if we want

to take more than an instrumentalist view of mental states. It then becomes crucial to say what implementation is and when a particular level of detail is part of a theory and when it is irrelevant 'mere' implementation. If, as Clark advocates (p. 141),

> cognitive science is an investigation of a mind composed of many interrelating virtual machines with correct psychological models at each level and further accounts required for the interrelationships between such levels,

teasing out a rigorous theory of algorithms at different levels of abstraction and implementation is a necessary ingredient of any well-founded cognitive science.

MORE IMPLICATIONS FOR EXPLANATION

The multilevel framework itself, in a context and with respect to a particular question, illustrates a kind of explanation. To borrow an example from Clark (1989, p. 129), suppose we have a system that computes the results of multiplying integers and we ask, 'How does it compute "8 x 7"?' Another way of stating the question is, how is the following sequence implemented?

input1: 8	input2: 7	output: U
input1: 8	input2: 7	output: 56

The level of detail to which we must descend and the cases to be considered depend on the larger context -- why do we want to know and how much do we want to know, from what angle? Three possible answers in terms of algorithmic sequences follow.

```
(1)  input1: 8   input2: 7    temp: U   output: U
     input1: 8   input2: 7    temp: 7   output: U
     input1: 8   input2: 7    temp: 14  output: U
     input1: 8   input2: 7    temp: 21  output: U
     input1: 8   input2: 7    temp: 28  output: U
     input1: 8   input2: 7    temp: 35  output: U
     input1: 8   input2: 7    temp: 42  output: U
     input1: 8   input2: 7    temp: 49  output: U
     input1: 8   input2: 7    temp: 56  output: U
     input1: 8   input2: 7    temp: 56  output: 56

(2)  input1: 8   input2: 7    temp: U   output: U
     input1: 8   input2: 7    temp: 49  output: U
     input1: 8   input2: 7    temp: 56  output: U
     input1: 8   input2: 7    temp: 56  output: 56

(3)  input1: 8   input2: 7    temp: U   output: U
     input1: 8   input2: 7    temp: 64  output: U
     input1: 8   input2: 7    temp: 56  output: U
     input1: 8   input2: 7    temp: 56  output: 56
```

The second sequence is a simple (selection of states) abstraction of the first, but it is also consistent with an explanation in terms of a table lookup for '7 x 7', followed by an addition of '7'. The third algorithmic sequence suggests a variation on this; it is consistent with a table lookup for '8 x 8', followed by a subtraction of '8'. A more detailed description of either might include the entire table as a value or values in each state.

The multilevel approach can be used to demonstrate the dangers of top-down theorising at this stage. Even if we do know that there is a more detailed implementation of the second sequence in terms of a lookup table, it is premature to conclude that

> The machine that stores the answer to '7 x 7' and adds 7 is in all probability faster. And if through damage it lost its capacity to add, it would still know the answer to '7 x 7' at least, whereas its more conventional cousin would not. (Clark, 1989, p. 129).

It could be that the table access and/or the addition are ultimately realised in very slow hardware. It could even be that the table access itself involves addition. What appears to be stored or data or explicit at one level is guaranteed to be distributed at another level under some description. A table may even be generated anew for each access or, at some level, there may be no table separate from the access rules. The point is that without bottom-up constraints as well, no conclusions about absolute time or physical damage can be drawn. If times can be associated with the steps of the given algorithms from measurements taken from running the actual system, and if these show fixed short times for the table lookup steps, then the speed claims are more reasonable. (Still, since we are dealing with abstractions, longer times for the addition may be complicated by details that are not apparent at a given level.) If there is evidence that addition always takes longer than the theoretical table lookups, for example, then there is more reason to claim that such lookups do not involve addition and may be immune to damage to addition.

A related point is the difficulty of separating out explicit from implicit representations, and process or

programs from data, as all of these are level-dependent notions. A so-called list in a LISP program may be distributed across internal storage and disc space at any given moment under a more detailed description. Data structures are often mentioned as examples of explicit representations, as in Clark and Karmiloff-Smith (forthcoming, p. 6):

> At level E-1 a redescription of the lower level has taken place such that knowledge embedded in the procedure is now available to the system as data. It is now explicitly defined in the internal representations...

But data are no more immune to level dependencies than anything else. And the situation may be complicated as we have seen, if they are explicit in a program that is compiled before it is run. If data compression and virtual memory are used and the compiled program and data are brought in and out of memory in pieces and put into different locations each time, good old solid explicit data structures begin to exhibit the characteristic features of pure distributed processing.

Clark and Karmiloff-Smith distinguish at least three stages of implicit and explicit knowledge in a system. The main distinguishing feature is the availability of knowledge. At the first stage, it is only available to a particular process, while at other stages it is available to other processes or to conscious access. The distinction between the implicit knowledge of the first stage and the sort of explicit knowledge of the second stage is defined in terms of data and procedures. The authors say of implicit knowledge that

> The model hypothesizes that knowledge stored as part of an effective procedure is independent of knowledge stored in other procedures. Thus, until the knowledge is redescribed and explicitly represented ..., identical knowledge

> components could be stored in different procedures without the system knowing this (p. 6).

And after redescription there is level E-1 where, as just quoted, '... knowledge embedded in the procedure is now available to the system as data'.

Some potential difficulties of placing too much weight on representational formats can be seen by extending a simple computational example (Clark and Karmiloff-Smith, forthcoming, p. 18). A planning system that uses a stack to encode a series of operations is contrasted to one that uses a list. Of the stack system they say 'the system is unable to proceed from operation 1 to operation 3 without performing operation 2 en route'. On the other hand, the list-based system has 'facilities for moving at will between items in the list. This representation overcomes the serial order constraint intrinsic to the pushdown stack'. In the algorithmic framework, the primary point of interest is the flexibility of output in the list system as opposed to the rigidity of the stack system. A suitable level of description might include algorithms (or sets) that only indicate the output operations and their order as in

STACK = the algorithm containing the sequence:

output: op1

output: op2

output: op3

LIST = the algorithm containing the following sequences (different sequences are separated by horizontal lines):

output: op1

output: op2

output: op3

---

output: op1

output: op3

output: op2

---

output: op2

output: op1

output: op3

---

output: op2

output: op3

output: op1

---

output: op3

output: op1

output: op2

---

output: op3

output: op2

output: op1

---

output: op1

output: op2

---

output: op1

output: op3

---

output: op2

output: op1

---

output: op2

output: op3

---

output: op3

output: op1

---

output: op3

output: op2

---

output: op1

---

output: op2

---

output: op3

The first algorithmic sequence in STACK can be shown to include a stack at a lower level of detail, following the verbal description. Using a typical stack description in terms of top and bottom pointers, a likely implementation of the sequence in STACK is:

```
output: U
top: 1          bottom: 3
stack: <(1, op1), (2, op2), (3, op3)>

output: op1
top: 2          bottom: 3
stack: <(1, op1), (2, op2), (3, op3)>

output: op2
top: 3          bottom: 3
stack: <(1, op1), (2, op2), (3, op3)>

output: op3
top: 0          bottom: 0
stack: <(1, op1), (2, op2), (3, op3)>
```

The stack value contains three items in 'addresses' 1, 2 and 3. The top of the stack is at address 1 and the bottom is at address 3, initially. When the top item in the stack (op1) is output, the top pointer is moved to 2, and so on. Using a typical list description in terms of head and tail pointers, an implementation of the same sequence from LIST could be given as follows:

```
output: U
head: 1   tail: 3
list: <(1, 0, op1, 2), (2, 1, op2, 3), (3, 2, op3, 0)>

output: op1
head: 1   tail: 3
list: <(1, 0, op1, 2), (2, 1, op2, 3), (3, 2, op3, 0)>

output: op2
head: 1   tail: 3
list: <(1, 0, op1, 2), (2, 1, op2, 3), (3, 2, op3, 0)>
```

output: op3
head: 1 tail: 3
list: ⟨(1, 0, op1, 2), (2, 1, op2, 3), (3, 2, op3, 0)⟩

The list value contains three items, again addressed by 1, 2 and 3. In addition, each element of the list value includes a pointer or address of its predecessor and successor in the list. The list item (1, 0, op1, 2) can be interpreted as: the item at address 1 is op1, the preceding item in the list is at address 0 (i.e., there is no preceding item, and item 1 is the head of the list) and the succeeding item is at address 2.

Even disregarding the fact that each operation in the stack system need not be performed when and only when it is removed from the stack, it is a mistake to conclude that the representations in STACK, or even any lower level representations on which they supervene, must change in order to be describable by LIST. For example it is possible that a valid lower level description of STACK may turn out to justify the assertion that the stack is implemented as a list, with the stack value ⟨(1, op1), (2, op2), (3, op3)⟩,
e.g., being implemented in the list value
⟨(1, 0, op1, 2), (2, 1, op2, 3), (3, 2, op3, 0)⟩
by the obvious functional relationship.

There are two possible accounts of a change or redescription from STACK to LIST (ignoring for simplicity any combinations of the possibilities). One is a shift of perspective by the theorist, but let us assume that the differences are deeper than that and that some physical change has taken place. This second possibility, that of physical change, can take place in a number of ways. The parts of the physical system that underlie the algorithmic description can themselves change. This would be the case if in the above example,

STACK were not at first describable at any lower level as a list system. At a later time, the same system may be changed so that it is describable as a list system, but there is no valid abstraction that meets all constraints and includes a top and bottom pointer (so it is no longer a stack). This would be a destructive modification. It is also possible that the system could change in such a way that both abstractions would remain valid, retaining features describable as top and bottom pointers and also features describable as head and tail pointers. Furthermore, it is possible that the physical system may change in such a way that the components underlying the original algorithm do not participate in the change. This is what Clark and Karmiloff-Smith appear to have in mind for their conservative (with respect to old representations) theory, in which

> To become a data structure available to other parts of the ... system outside the particular content-bound, special-purpose procedure, the knowledge has to be redescribed into an accessible format. (p. 8).

The algorithmic framework gives us a way to consider two ways in which this might happen without having to decide what is data and what is process across levels. The most likely equivalent of what Clark and Karmiloff-Smith mean, as I understand them, is that, while leaving in place the physical underpinnings of the stack algorithms, a change occurs which results in the system being describable in terms of lists as well. The talk of data structures leads me to think of this as some sort of copy of the implicit stack, now available to other procedures as a flexible list. Another possibility is that there is no such copy but that a change in the system has now allowed access by modified or new procedures to the very same 'implicit' data structure. Either of these cases would reveal the list as an explicit representation,

defined simply as a labelled value in an appropriate algorithm at some level of description in the multilevel framework.

A more straightforward example is the well-worn virtual governor example from Dewan (1976), described by Hooker (1981), and repeated by P.S. Churchland (1986). Given the flexibility that must be built into multilevel representations if they are to cope with even ordinary computer systems, I find it compelling in advocating a generic framework for multilevel descriptions in terms of states. Here is the situation (Churchland, 1986, p. 365):

> Consider a set of electrical generators G, each of which produces alternating current electrical power at 60 Hz but with fluctuations in frequency of 10% around some average value. Taken singly, the frequency variability of the generators is 10%. Taken joined together in a suitable network, their collective frequency variability is only a fraction of that figure because, statistically, generators momentarily fluctuating behind the average output in phase are compensated for by the remaining generators, and conversely, generators momentarily ahead in phase have their energy absorbed by the remainder. The entire system functions, from an input/output point of view, as a single generator with a greatly increased frequency reliability, or, as control engineers express it, with a single, more powerful, 'virtual governor'. The property 'has a virtual governor of reliability f' is a property of the system as a whole, but of none of its components.

The virtual governor is just distributed physically, as a perfectly acceptable 'explicit' data structure might be. There is a respectable function mapping the combined powers of the individual generators at one level of description to the power of the virtual governor. Similarly, the power of the individual governors is

describable as a function of the interaction of their components, so they are abstract or virtual as well.

As a final point, the nature of programme and process explanations (as defined by Jackson and Pettit, 1988, and discussed by Clark, 1989) can be elucidated by the algorithmic approach. Of course, I balk at the terminology which distinguishes programme and process, seeing the distinction rather as one of more or less abstract algorithms (with respect to detail). Some examples are given by Jackson and Pettit (1988, p. 392 - 393) and repeated by Clark (1989, p. 197).

> Electrons A and B are acted on by independent forces F_A and F_B respectively, and electron A then accelerates at the same rate as electron B. The explanation of this fact is that the magnitude of the two forces is the same ... But this sameness in magnitude is quite invisible to A ... This sameness does not make A move off more or less briskly; what determines the rate at which A accelerates is the magnitude of F_A, not that magnitude's relationship to another force altogether.

They go on to say (p. 393 - 394) that

> We can express the basic idea behind a programme explanation in terms of what remains constant under variation. Suppose state A caused state B. Variations on A, say, A', A", ... would have caused variations on B, say, B', B", ..., respectively. It may be that if the A^i share a property P, the B^i would share a property Q: keep P constant among the actual and possible causes, and Q remains constant among the actual and possible effects. If you like, Q tracks P. Our point is that in such a case P causally explains Q by programming it even though it may be that P does not produce Q.

In algorithmic terms, the example could be described as a state containing values for the acceleration of A, the acceleration of B and the forces F_A and F_B, as in

acceleration rate of A: v1
acceleration rate of B: v2
F_A: v3
F_B: v3

A more detailed description could include movement of A and B, and a causal story could be told at either level relating F_A to A and F_B to B. Nonetheless, the four-value state is a reasonable rendition of the level at which the problem is stated. The explanation suggested by Jackson and Pettit can be seen as an abstraction. F_B and F_A are combined and their values subtracted. The level of description of the explanation can be given as a state containing values for the acceleration of A, the acceleration of B and the magnitude of the difference of F_A and F_B, which will be close to zero when A and B's accelerations are close together:

acceleration rate of A: v1
acceleration rate of B: v2
$|F_A - F_B|$: 0

The fact that 'sameness in magnitude is quite invisible to A' is reflected in the descriptions being given for a system containing more than just A. If A itself somehow recorded or remembered B's acceleration, then there may have been an explanatory algorithm involving levels of description of A alone.

There is nothing special about the particular levels employed or of the particular context in which they are employed. There is no inherent 'programme' or 'process' level; all there is is a two-level description where one level has been described as an abstraction and one as an implementation relative to the other. Either of these

levels of detail could be abstracted in other ways or implemented in other ways.

The new multilevel framework provides a way to relate various system descriptions at various levels of detail, constrained by bottom-up measurements and top-down theorising and perhaps independently motivated restrictions on abstraction functions. It provides a rigorous, formal foundation for claims that systems can share a property at one level and yet be radically different at another, without prejudging which level is right or more appropriate in a given context. Ignoring development, semantics and causation for the most part, this relatively simple state-based approach can be stretched surprisingly far. In contrast to virtual machines or instrumentalist explanations, it retains throughout the notions of strong equivalence between algorithms and well-defined supervenience of states at a given level of description on substates at lower levels of detail.

CHAPTER VII

CONCLUSION

In the massively multilevel theory of strong equivalence of complex systems, answers can be given to the questions raised in the introduction: What is an algorithm? When are two algorithms the same?

An algorithm is a finite set of finite sequences of finite states, defining an ideal machine and a level of description. Two algorithms are the same when they are equal by the usual set equality. Given the definitions of abstraction and implementation, we might also say that two algorithms are the same in a different less precise sense if one is an abstraction of the other. In that case, they are the same when viewed at the level of description defined by the more abstract set; the more detailed version is defined as a valid implementation.

The framework resulting from interdependent definitions of levels of detail, algorithms, algorithmic sequences, states, ideal machines, abstraction and implementation provides a foundation for all these terms as they are used in cognitive science and much of computer science. So when we speak of connectionist algorithms for pattern recognition, we can underwrite it with sequences of states of activation values for example. If the intended level of theoretical interest of a system realising such an algorithm is more or less detailed than that, it can still be specified within the framework. When students of programming are asked to implement a particular algorithm such as 'binary search' or 'heap sort' in some formal language, we can see it as an assignment to write a program that is an abbreviation for a set of state sequences, which is itself a more detailed version of the set of state sequences implied by the

given textbook description (very likely in procedural form) of binary search or heap sort.

Most importantly, we now have the means to compare and contrast algorithms directly, across the boundaries of particular languages, architectures and hardware realisations. From this vantage point, individual connectionist and classical models can be compared in new ways. For instance, the claim that a connectionist model is an implementation of a classical one can be stated formally in terms of the level of description at which each model is intended. Their relationship in terms of abstraction and implementation can then be stated as a theorem and proved, if indeed it is true. On the other hand, no defining criteria emerge for distinguishing equivalence classes of connectionist or classical algorithms. Typical characteristics of connectionist systems at their usual levels of description are seen to be a large number of different paths from any input to its associated output(s) and a large number of different values changing with any single state change. Such distinctions are overshadowed, however, by the similarities of different types of models when viewed at multiple levels of description.

The recognition of a systematic ambiguity between concreteness and detail in current usage of 'implementation' and 'abstraction' motivated clarification and a natural extension of Marr's levels. The resultant closer look at algorithms at different levels of detail led to the new definition of algorithms in terms of state sequences. These sets of possible state sequences can be seen as ideal machines, since all attainable states and all (possibly context-dependent) transitions are given. They can be seen to extend Turing's ideal machine in a number of ways. First, there is not just one sort of machine with one set of

operations and one set of states. Instead, a complex system can be described by an arbitrarily large number of ideal machines, related by abstraction and implementation as defined. Turing's 'states of mind' (Turing, 1937) or machine states are still finite, consistent with his original exposition, but they can be described from infinitely many perspectives. Infinity still comes into the framework as a result, but not in any way analagous to Turing's infinite tape. Storage and input values are bounded, identifying a subset of partially computable functions as 'bounded' computable functions. This restriction is in keeping with Turing's arguments for finitude elsewhere in his theory on the basis of human limitations. For example, the symbol set is finite as is the set of internal states or 'states of mind':

> The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be 'arbitrarily close' and will be confused. (Turing, 1937, p. 136 in Davis, 1965).

These arguments were not carried over to the tape, perhaps because Turing saw the tape as part of the environment, outside the human system. (This appears to be the case according to his biographer; see Hodges, 1983). The Turing machine was developed as a tool in the search for an unsolvable problem, which in Turing's framework amounted to an infinitely long number for which there was no possible 'definite procedure' to print the nth digit. For this project, an unbounded tape was essential. Limiting the input values is a restriction on Turing's formulation in one way; in another it is an extension of his appeal to finiteness, applied to cases where the tape is considered as part of the system rather than as something outside it. Furthermore, the Turing machine as originally defined was deterministic. That is, the internal state and the current symbol being

scanned completely determined the next state. Later, nondeterministic Turing machines were defined and shown to be equivalent to deterministic Turing machines (for a proof of this, see any standard theoretical computer science text, e.g. Davis and Weyuker, 1983). The new framework, with its many levels, shows how the same system might be described as deterministic at one level and nondeterministic at other -- possibly higher, possibly lower, possibly even both -- levels. Finally, Turing-machine equivalence is weak equivalence in contrast to the new theory's strong equivalence. Two procedures are Turing-machine equivalent if there exists a Turing machine, any Turing machine, that will calculate the same output from the same input. The same output can include being undefined, or not halting, in the case of functions that are not computable. Somewhat surprisingly, therefore, the class of algorithms may not be a straightforward subset of the class of partially computable functions. Inputs that are undefined at one level of description may or may not cause the system to halt, depending on more detailed implementations.

Of course there is no claim that the multilevel framework as defined here is the last word. Even Turing's machine as originally defined was corrected (see the discussion in Davis, 1965, and in Post, 1948, reprinted in the same volume), and its implications are still being considered. Ideally, the new theory will be refined and widely applied. It has in common with Turing's machine definiteness (in the sense of rigour rather than determinacy) combined with a relatively simple and intuitive (perhaps operational) definition. It has the potential to provide a common foundation to facilitate not only the inevitable further debate about appropriate levels of description and explanation in general, but also comparisons between particular theories, models, programs and systems.

Apart from the continued application and testing of the framework in these ways, I will mention three general directions for further investigation. On the purely philosophical side, there is room for better understanding of the role of finite, state-based descriptions and how they fit into the large and growing literature about explanation and observation.

On the computer science and software engineering side, the theory of states and algorithms at various levels of detail carries over quite directly into terms of design and development of complex systems. Confusion about levels, algorithms, abstraction and implementation in this area is remarkably similar to that in cognitive science. For example, software engineers tend to use a three-level system comprised of requirements, design and implementation, each with its own typical notation for system description. The data flow diagrams of requirements are frequently claimed to give the 'what' and not the (mere) 'how' of the lower levels which might be initially described in a flow chart (see, e.g., Fairley, 1985; Sommerville, 1982). Translation from one level to the next (one way of looking at the design problem) does not take into account any sort of strong equivalence, a major flaw in the foundations of software engineering. The new theory is just what is needed to begin to remedy this problem. In more practical terms, it is not clear that the cumbersome sets and states of the multilevel framework could themselves be turned into a new design methodology, but it is an area for continued investigation.

Between these two extremes, or perhaps combining them, there is room for further exploration of the nature of the abstraction/implementation relationship, no doubt leading to refinements of the abstraction operations or

constraints on them. Such work may well be enhanced by a computer model of the framework for some particular algorithms, allowing descriptions at different levels to be extracted and simulated, related by the functions that define the abstractions.

APPENDIX A

A QUICKSORT ALGORITHM BASED ON BROOKSHEAR'S PSEUDOCODE PROCEDURE

The labels in the algorithm and their intended interpretation are:

LIST: the current list of items to be sorted.
PIVOT: the item selected to partition the current list.
TOP: the pointer used to compare items, starting from the top (beginning) of the list, with the pivot.
BOTTOM: the position used to compare items with the pivot, starting from the bottom (end) of the list.
SUBLIST1: the sublist of the current list before the pivot.
SUBLIST2: the sublist of the current list after the pivot.
WHOLE LIST: the entire list, as opposed to the current (sub-) list.

This algorithm contains only one sequence, the one for the input list comprising JANE, BOB, ALICE, TOM, CAROL, BILL, GEORGE, CHERYL, SUE and JOHN. A new state is given for each 'call' to the Brookshear procedure (see Chapter IV), each initialisation of local variables (PIVOT, TOP and BOTTOM), each (composite) move of the bottom pointer, each (composite) move of the top pointer, each interchange of names or a name and the pivot, and each 'return' to a calling procedure. Changed values between two adjacent states are underlined in the second state, as an aid to comprehension. Comments before each state serve to associate the change with Brookshear's procedure.

```
Initial state.
LIST
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

Initialise pivot and pointers.
LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT:  JANE   TOP:  1   BOTTOM:  10
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

Move bottom pointer.
LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT:  JANE   TOP:  1   BOTTOM:  8
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
```

```
Move top pointer.
LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT:  JANE    TOP:  4    BOTTOM:  8
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

Interchange names indexed by pointers.
LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
PIVOT:  JANE    TOP:  4    BOTTOM:  8
SUBLIST1:
( )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

Move bottom pointer.
LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
PIVOT:  JANE    TOP:  4    BOTTOM:  7
SUBLIST1:
( )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
```

```
Move top pointer.
LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
PIVOT:  JANE   TOP:  7   BOTTOM:  7
SUBLIST1:
( )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

Interchange pivot and name indexed by pointers.
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
PIVOT:  JANE   TOP:  7   BOTTOM: 7
SUBLIST1:
(GEORGE BOB ALICE CHERYL CAROL BILL)
SUBLIST 2:
TOM SUE JOHN)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)

New initial state for sublist 1 above.
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT:  U   TOP: U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
```

```
Initialise pivot and pointers.
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT:  GEORGE    TOP:  1    BOTTOM:  6
SUBLIST1:
(__)
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)

Move bottom pointer.
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT:  GEORGE    TOP:  1    BOTTOM:  6
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)

Move top pointer.
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT:  GEORGE    TOP:  6    BOTTOM:  6
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
```

```
Interchange pivot and name indexed by pointer.
LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE)
PIVOT:  GEORGE   TOP:  6   BOTTOM:  6
SUBLIST1:
(BILL BOB ALICE CHERYL CAROL)
SUBLIST2:
(  )
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE TOM SUE JOHN)

New initial state for SUBLIST1 above.
LIST:
(BILL BOB ALICE CHERYL CAROL)
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Initialise pivot and pointer.
LIST:
(BILL BOB ALICE CHERYL CAROL)
PIVOT:  BILL   TOP:  1   BOTTOM:  5
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL)
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

```
Move bottom pointer.
LIST:
(BILL BOB ALICE CHERYL CAROL)
PIVOT:  BILL   TOP:  1   BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL)
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Move top pointer.
LIST:
(BILL BOB ALICE CHERYL CAROL)
PIVOT:  BILL   TOP:  2    BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL)
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Interchange names indexed by pointers.
LIST:
(BILL ALICE BOB CHERYL CAROL)
PIVOT:  BILL   TOP:  2   BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(ALICE BOB CHERYL CAROL)
WHOLE LIST:
(BILL ALICE BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

```
Move bottom pointer.
LIST:
(BILL ALICE BOB CHERYL CAROL)
PIVOT:  BILL   TOP:  2   BOTTOM:  2
SUBLIST1:
(  )
SUBLIST2:
(ALICE BOB CHERYL CAROL)
WHOLE LIST:
(BILL ALICE BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Move top pointer.
LIST:
(BILL ALICE BOB CHERYL CAROL)
PIVOT:  BILL   TOP:  2   BOTTOM:  2
SUBLIST1:
(  )
SUBLIST2:
(ALICE BOB CHERYL CAROL)
WHOLE LIST:
(BILL ALICE BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Interchange pivot and name indexed by pointers.
LIST:
(ALICE BILL BOB CHERYL CAROL)
PIVOT:  BILL   TOP:  2   BOTTOM:  2
SUBLIST1:
(ALICE)
SUBLIST2:
(BOB CHERYL CAROL)
WHOLE LIST
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

```
New initial state (trivially sorted).
LIST:
(ALICE)
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Return.
LIST:
(ALICE BILL BOB CHERYL CAROL)
PIVOT:  BILL   TOP:  2   BOTTOM:  2
SUBLIST1:
(ALICE)
SUBLIST2:
(BOB CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

New initial state for SUBLIST2 above.
LIST:
(BOB CHERYL CAROL)
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

```
Initialise pivot and pointers.
LIST:
(BOB CHERYL CAROL)
PIVOT:  BOB   TOP:  1   BOTTOM:  3
SUBLIST1:
(__)
SUBLIST2:
(CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Move bottom pointer.
LIST:
(BOB CHERYL CAROL)
PIVOT:  BOB   TOP:  1   BOTTOM:  1
SUBLIST1:
(  )
SUBLIST2:
(CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Move top pointer.
LIST:
(BOB CHERYL CAROL)
PIVOT:  BOB   TOP:  1   BOTTOM:  1
SUBLIST1:
(  )
SUBLIST2:
(CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

```
Interchange pivot and name indexed by pointers.
LIST:
(BOB CHERYL CAROL)
PIVOT:  BOB   TOP:  1   BOTTOM:  1
SUBLIST1:
(  )
SUBLIST2:
(CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

New initial state for SUBLIST1 above (trivially sorted).
LIST:
( )
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Return.
LIST:
(BOB CHERYL CAROL)
PIVOT:  BOB   TOP:  1  BOTTOM:  1
SUBLIST1:
(  )
SUBLIST2:
(CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

```
New initial state for SUBLIST2 above.
LIST:
(CHERYL CAROL)
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Initialise pivot and pointers.
LIST:
(CHERYL CAROL)
PIVOT:  CHERYL  TOP:  1   BOTTOM:  2
SUBLIST1:
(  )
SUBLIST2:
(CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Move bottom pointer.
LIST:
(CHERYL CAROL)
PIVOT:  CHERYL  TOP:  1   BOTTOM:  2
SUBLIST1:
(  )
SUBLIST2:
(CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)
```

```
Move top pointer.
LIST:
(CHERYL CAROL)
PIVOT:  CHERYL  TOP:  2   BOTTOM:  2
SUBLIST1:
(  )
SUBLIST2:
(CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE TOM SUE JOHN)

Interchange pivot and name indexed by pointers.
LIST:
(CAROL CHERYL)
PIVOT:  CHERYL   TOP:  2   BOTTOM:  2
SUBLIST1:
(CAROL)
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)

New initial state for SUBLIST1 above (trivially sorted).
LIST:
(CAROL)
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)
```

```
Return.
LIST:
(CAROL CHERYL)
PIVOT:  CHERYL   TOP:  2   BOTTOM:  2
SUBLIST1:
(CAROL)
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)

New initial state for SUBLIST2 above (trivially sorted).
LIST:  (  )
PIVOT:  U  TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)

Return.
LIST:
(CAROL CHERYL)
PIVOT:  CHERYL   TOP:  2   BOTTOM:  2
SUBLIST1:
(CAROL)
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)
```

```
Return.
LIST:
(BOB CAROL CHERYL)
PIVOT:  BOB   TOP:  1   BOTTOM:  1
SUBLIST1:
(  )
SUBLIST2:
(CAROL CHERYL)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)

Return.
LIST:
(ALICE BILL BOB CAROL CHERYL)
PIVOT:  BILL   TOP:  2   BOTTOM:  2
SUBLIST1:
(ALICE)
SUBLIST2:
(BOB CAROL CHERYL)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)

Return.
LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE)
PIVOT:  GEORGE  TOP:  6   BOTTOM:  6
SUBLIST1:
(ALICE BILL BOB CAROL CHERYL)
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)
```

```
New initial state for SUBLIST2 above (trivially sorted).
LIST:
(  )
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)

Return.
LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE)
PIVOT:  GEORGE   TOP:  6   BOTTOM:  6
SUBLIST1:
(ALICE BILL BOB CAROL CHERYL)
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)

Return.
LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)
PIVOT:  JANE   TOP:  7   BOTTOM:  7
SUBLIST1:
(ALICE BILL BOB CAROL CHERYL GEORGE)
SUBLIST2:
(TOM SUE JOHN)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)
```

```
New initial state for SUBLIST2 above.
LIST:
(TOM SUE JOHN)
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)

Initialise pivot and pointers.
LIST:
(TOM SUE JOHN)
PIVOT:  TOM   TOP:  1   BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(SUE JOHN)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)

Move bottom pointer.
LIST:
(TOM SUE JOHN)
PIVOT:  TOM   TOP:  1   BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(SUE JOHN)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)
```

```
Move top pointer.
LIST:
(TOM SUE JOHN)
PIVOT:  TOM   TOP:  3  BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(SUE JOHN)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE TOM SUE JOHN)

Interchange pivot and name indexed by pointers.
LIST:
(JOHN SUE TOM)
PIVOT:  TOM   TOP:  3   BOTTOM:  3
SUBLIST1:
(JOHN SUE )
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

New initial state for SUBLIST1 above.
LIST:
(JOHN SUE)
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)
```

```
Initialise pivot and pointers.
LIST:
(JOHN SUE)
PIVOT:  JOHN   TOP:  1  BOTTOM:  2
SUBLIST1:
(  )
SUBLIST2:
(SUE)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

Move bottom pointer.
LIST:
(JOHN SUE)
PIVOT:  JOHN   TOP:  1  BOTTOM:  1
SUBLIST1:
(  )
SUBLIST2:
(SUE)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

Move top pointer.
LIST:
(JOHN SUE)
PIVOT:  JOHN   TOP:  1  BOTTOM:  1
SUBLIST1:
(  )
SUBLIST2:
(SUE)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)
```

```
Interchange pivot and name indexed by pointers.
LIST:
(JOHN SUE)
PIVOT:  JOHN   TOP:  1  BOTTOM:  1
SUBLIST1:
(  )
SUBLIST2:
(SUE)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

New initial state for SUBLIST1 above (trivially sorted).
LIST:
(  )
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST12:
U
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

Return.
LIST:
(JOHN SUE)
PIVOT: JOHN   TOP:  1   BOTTOM:  1
SUBLIST1:
(  )
SUBLIST2:
(SUE)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)
```

```
New initial state for SUBLIST2 above (trivially sorted).
LIST:
(SUE)
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

Return.
LIST:
(JOHN SUE)
PIVOT:  JOHN   TOP:  1   BOTTOM:  1
SUBLIST1:
(  )
SUBLIST2:
(SUE)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

Return.
LIST:
(JOHN SUE TOM)
PIVOT:  TOM   TOP:  3   BOTTOM:  3
SUBLIST1:
(JOHN SUE)
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)
```

```
New initial state for SUBLIST2 above (trivially sorted).
LIST:
(  )
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

Return.
LIST:
(JOHN SUE TOM)
PIVOT:  TOM   TOP:  3   BOTTOM:  3
SUBLIST1:
(JOHN SUE)
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

Return.
LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)
PIVOT:  JANE   TOP:  7 BOTTOM:  7
SUBLIST1:
(ALICE BILL BOB CAROL CHERYL GEORGE)
SUBLIST2:
(JOHN SUE TOM)
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

End.
```

APPENDIX B

A QUICKSORT ALGORITHM BASED ON KNUTH'S MIX PROGRAM

The labels in the algorithm and their intended interpretation are:

INPUT: the list of items to be sorted, preceded by -INF(inity) and followed by +INF(inity).

N: the number of items in the original INPUT.

M: the minimum list size; if M = 1 then lists of 1 or fewer items are not sorted by the Quicksort method.

STACK: explicit stack for handling recursion.

rI1: used in conjunction with moves to hold locations; miscellaneous integer values.

rI2: the position in INPUT of the first item in the list currently being sorted; Knuth also calls this 'l' (for left).

rI3: the position in INPUT of the last item in the list currently being sorted; Knuth also calls this 'r' (for right).

rI4: the pointer used to compare items, starting from the left (beginning) of the list, with the pivot; Knuth also calls this 'i' It is close to 'TOP' in Appendix A.

rI5: the pointer used to compare items, starting from the right (end) of the list, with the pivot; Knuth also calls this 'j'. It is close to 'BOTTOM' in Appendix A.

rI6: the size of the stack.

rA: the pivot; Knuth calls this 'K' for key.

rX: miscellaneous integer and other values; used in exchanges, for example.

This algorithm, like the one in Appendix A, contains only one sequence, again the one for the input list containing

JANE, BOB, ALICE, TOM, CAROL, BILL, GEORGE, CHERYL, SUE and JOHN. The values (for all label-value pairs except the INPUT, the STACK, N and M) are displayed in rows with column headings to indicate the labels. Only those values that have changed from the previous state are displayed, usually. Each row represents a new state. For changes to the input/output list or to the stack, the entire list or stack will be printed across the columns with the changed items underlined. No other values change at the same time, so no information will be lost. N and M remain constant throughout in this example, with N = 10 and M = 1. Comments appear at the right to associate the state change with Knuth's MIX program. Detailed explanations of the MIX assembly language and object language are given in Volume 1 of Knuth's The Art of Computer Programming (1968).

```
rI1       rI2  rI3  rI4  rI5  rI6  rA   rX   Comments
          (l)  (r) (TOP)(BOT)   (PIVOT)
-------------------------------------------------------
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
                                0             ENT6 0
          1                                   ENT2 1
               10                             ENT3 N
                         11                   ENT5 1,3
                                    JANE       LDA INPUT,2
                    2                          ENT4 1,2
                    3                          INC4 1
                    4                          INC4 1
                         10                    DEC5 1
                          9                    DEC5 1
                          8                    DEC5 1
                                          8    ENTX 0,5
                                          4    DECX 0,4
                                         TOM   LDX INPUT,4
INPUT+4                                        ENT1 INPUT,4

(JANE BOB ALICE CHERYL CAROL BILL GEORGE CHERYL SUE JOHN)
```

```
rI1        rI2  rI3  rI4  rI5  rI6  rA  rX   Comments
           (1)  (r) (TOP)(BOT)   (PIVOT)
-------------------------------------------------------
                                          MOVE INPUT,5

(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

                                          STX INPUT,5
                    5                     INC4 1
                    6                     INC4 1
                    7                     INC4 1
                    8                     INC4 1
                         7                DEC5 1
                                      7   ENTX 0,5
                                     -1   DECX 0,4
                                   GEORGE LDX INPUT,5

(GEORGE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)

                    10                    ENT4 0,3
                    2                     DEC4 M,5
7                                         ENT1 0,5
5                                         DEC1 M,2
                                      2   ENTA 0,4
                                     -3   DECA 0,1
                              1           INC6 1
          STACK: 1U                       ST2 STACK,6(A)
                                      6   ENTA -1,5
          STACK: 16                       STA STACK,6(B)
8                                         ENT2 1,5
                         11               ENT5 1,3
                                   TOM    LDA INPUT,2
               9                          ENT4 1,2
               10                         INC4 1
               11                         INC4 1
```

```
rI1        rI2  rI3  rI4  rI5  rI6  rA  rX   Comments
           (l)  (r) (TOP)(BOT)   (PIVOT)
-------------------------------------------------------
                          10                DEC5 1
                                        10  ENTX 0,5
                                        -1  DECX 0,4
                                       JOHN LDX INPUT,5

(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE JOHN)

(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)

                     10                   ENT4 0,3
                     -1                   DEC4 M,5
10                                        ENT1 0,5
1                                         DEC1 M,2
                                  -1      ENTA 0,4
                                  -2      DECA 0,1
                9                         ENT3 -1,5
                      10                  ENT5 1,3
                                  JOHN    LDA INPUT,2
                     9                    ENT4 1,2
                          9               DEC5 1
                          8               DEC5 1
                                      8   ENTX 0,5
                                     -1   DECX 0,4
                                    JOHN  LDX INPUT,5

(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)

                                          STX INPUT,2

(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)

                                          STA INPUT,5
                     9                    ENT4 0,3
                     0                    DEC4 M,5
8                                         ENT1 0,5
```

```
rI1        rI2  rI3  rI4  rI5  rI6  rA   rX  Comments
           (l)  (r) (TOP)(BOT)   (PIVOT)
-------------------------------------------------------
-1                                          DEC1 M,2
                                       0    ENTA 0,4
                                       1    DECA 0,1
           1                                LD2 STACK,6(A)
                6                           LD3 STACK,6(B)
                     0                      DEC6 1
           7                                ENT5 1,3
                          GEORGE            LDA INPUT,2
      2                                     ENT4 1,2
      3                                     INC4 1
      4                                     INC4 1
      5                                     INC4 1
      6                                     INC4 1
      7                                     INC4 1
           6                                DEC5 1
                                       6    ENTX 0,5
                                      -1    DECX 0,4
                                    BILL    LDX INPUT,5

(BILL BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)

                                            STX INPUT,2

(BILL BOB ALICE CHERYL CAROL GEORGE JANE JOHN SUE TOM)

                                            STA INPUT,5
      6                                     ENT4 0,3
      -1                                    DEC4 M,5
      6                                     ENT1 0,5
      4                                     DEC1 M,2
                                       -1   ENTA 0,4
                                       -5   DECA 0,1
                5                           ENT3 -1,5
                     6                      ENT5 1,3
                                    BILL    LDA INPUT,2
```

```
rI1        rI2  rI3  rI4   rI5   rI6   rA   rX    Comments
           (l)  (r) (TOP)(BOT)    (PIVOT)
-------------------------------------------------------------
                     2                          ENT4 1,2
                          5                     DEC5 1
                          4                     DEC5 1
                          3                     DEC5 1
                                           3    ENTX 0,5
                                           1    DECX 0,4
                                          BOB   LDX INPUT, 4
INPUT+2                                         ENT1 INPUT,4

(BILL ALICE ALICE CHERYL CAROL GEORGE JANE JOHN SUE TOM)

                                                MOVE INPUT,5

(BILL ALICE BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

                                                STX INPUT,5
                               3                INC4 1
                                    2           DEC5 1
                                           2    ENTX 0,5
                                          -1    DECX 0,4
                                        ALICE   LDX INPUT,5

(ALICE ALICE BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

                                                STX INPUT,2

(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

                                                STA INPUT,5
                               5                ENT4 0,3
                               2                DEC4 M,5
2                                               ENT1 0,5
0                                               DEC1 M,2
                                      2         ENTA 0,4
                                      2         DECA 0,1
```

```
rI1        rI2  rI3  rI4  rI5  rI6  rA  rX   Comments
           (l)  (r) (TOP)(BOT)   (PIVOT)
-------------------------------------------------------
            3                                ENT2 1,5
                          6                  ENT5 1,3
                                       BOB   LDA INPUT,2
                     4                       ENT4 1,2
                          5                  DEC5 1
                          4                  DEC5 1
                          3                  DEC5 1
                                         3    ENTX 0,5
                                        -1    DECX 0,4
                                       BOB    LDX INPUT,5

(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

                                              STX INPUT,2

(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

                                              STA INPUT,5
                     5                        ENT4 0,3
                     1                        DEC4 M,5
3                                             ENT1 0,5
-1                                            DEC1 M,4
                                         1    ENTA 0,4
                                         2    DECA 0,1
            4                                 ENT2 1,5
                          6                   ENT5 A,3
                                     CHERYL   LDA INPUT,2
                     5                        ENT4 1,2
                     6                        INC4 1
                          5                   DEC5 1
                                         5    ENTX 0,5
                                        -1    DECX 0,4
                                     CAROL    LDX INPUT,5

(ALICE BILL BOB CAROL CAROL GEORGE JANE JOHN SUE TOM)
```

```
rI1        rI2  rI3  rI4  rI5  rI6  rA  rX  Comments
           (l)  (r) (TOP)(BOT)   (PIVOT)
-------------------------------------------------------

                                          STX INPUT,2

(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

                                          STA INPUT,5
                    5                     ENT4 0,3
                    -1                    DEC4 M,5
5                                         ENT1 0,5
0                                         DEC1 M,2
                                    -1    ENTA 0,4
                                    -1    DECA 0,1
          1                               LD2 STACK,6(A)
               6                          LD3 STACK,6(B)
                              -1          DEC6 1
```

At this point the stack is empty, and the algorithm is finished. Since M = 1, I have not included states for the final sort which could be used for cases when M is greater than 1.

APPENDIX C

A QUICKSORT ALGORITHM ABSTRACTED FROM AN ALGORITHM BASED ON KNUTH'S MIX PROGRAM

The labels in the algorithm and their intended interpretation are:

LIST: the current list of items to be sorted.
PIVOT: the item selected to partition the current list.
TOP: the pointer used to compare items, starting from the top (beginning) of the list, with the pivot.
BOTTOM: the pointer used to compare items with the pivot, starting from the bottom (end) of the list.
SUBLIST1: the sublist of the current list before the pivot.
SUBLIST2: the sublist of the current list after the pivot.
WHOLE LIST: the entire list, as opposed to the current (sub-) list.

As in Appendix A and Appendix B, this algorithm contains only one sequence, the one for the input list JANE, BOB, ALICE, TOM, CAROL, BILL, GEORGE, CHERYL, SUE and JOHN. It has been abstracted from the algorithm of Appendix B in a way that is meant to be suggestive of and bring it into more direct comparison with the algorithm of Appendix A. The following abstraction operations were used. The (unlabelled) list of Appendix B is now called WHOLE LIST. In this appendix, LIST, the current list being sorted, is a function of WHOLE LIST and rI2 (left, l) and rI3 (right, r) which give the boundary items of the current list in the more detailed version. PIVOT, TOP and BOTTOM are renamings of rA, rI4 and rI5,

respectively. SUBLIST1 and SUBLIST2 can be defined as functions of the current list (or the whole list and end points) and PIVOT. A new state is recorded for changes to these values, generally. A new state may be recorded as well for an interchange which leaves the list the same (an item is changed with itself), and only one cumulative state change is given for the incremental changes to the top and bottom pointers.

```
Initial state.
LIST
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT:  U   TOP:  U   BOTTOM:  U
SUBLIST1:
U
SUBLIST2:
U
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

Initialise PIVOT and pointers.
LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT:  JANE   TOP:  2   BOTTOM:  11
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
```

```
Move top pointer.
LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT:  JANE    TOP:  4    BOTTOM:  11
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

Move bottom pointer.
LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
PIVOT:  JANE    TOP:  4    BOTTOM:  8
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE TOM CAROL BILL GEORGE CHERYL SUE JOHN)

Interchange names indexed by pointers.
LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
PIVOT:  JANE    TOP:  4    BOTTOM:  8
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
```

```
Move top pointer.
LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
PIVOT:  JANE   TOP:  8   BOTTOM:  8
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

Move bottom pointer.
LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
PIVOT:  JANE   TOP:  8   BOTTOM:  7
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)
WHOLE LIST:
(JANE BOB ALICE CHERYL CAROL BILL GEORGE TOM SUE JOHN)

Interchange pivot and name indexed by bottom pointer.
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
PIVOT:  JANE   TOP:  8   BOTTOM:  7
SUBLIST1:
(GEORGE BOB ALICE CHERYL CAROL BILL)
SUBLIST2:
(TOM SUE JOHN)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
```

```
New initial state for SUBLIST2 above.
LIST:
(TOM SUE JOHN)
PIVOT:  JANE   TOP:  2   BOTTOM:  7
SUBLIST1:
( )
SUBLIST2:
(  )
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)

Initialise pivot and pointers.
LIST:
(TOM SUE JOHN)
PIVOT:  TOM   TOP:  9   BOTTOM:  11
SUBLIST1:
(  )
SUBLIST2:
(SUE JOHN)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)

Move top pointer.
LIST:
(TOM SUE JOHN)
PIVOT:  TOM   TOP:  11    BOTTOM:  11
SUBLIST1:
(  )
SUBLIST2:
(SUE JOHN)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)
```

```
Move bottom pointer.
LIST:
(TOM SUE JOHN)
PIVOT:  TOM   TOP:  11   BOTTOM:  10
SUBLIST1:
(  )
SUBLIST2:
(SUE JOHN)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE TOM SUE JOHN)

Interchange pivot and name indexed by bottom pointer.
LIST:
(JOHN SUE TOM)
PIVOT:  TOM   TOP:  9   BOTTOM:  10
SUBLIST1:
(JOHN SUE)
SUBLIST2:
(  )
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)

New initial state for SUBLIST1 above.
LIST:
(JOHN SUE)
PIVOT:  TOM   TOP:  1   BOTTOM:  11
SUBLIST1:
(  )
SUBLIST2:
(  )
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)
```

```
Initialise pivot and pointers.
LIST:
(JOHN SUE)
PIVOT: JOHN   TOP:  9  BOTTOM:  10
SUBLIST1:
( )
SUBLIST2:
(SUE)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)

Move bottom pointer.
LIST:
(JOHN SUE)
PIVOT: JOHN   TOP:  9  BOTTOM:  8
SUBLIST1:
(  )
SUBLIST2:
(SUE)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)

Interchange pivot and name indexed by bottom pointer.
LIST:
(JOHN SUE)
PIVOT:  JOHN   TOP:  9   BOTTOM: 8
SUBLIST1:
(  )
SUBLIST2:
(SUE)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)
```

```
New initial state for earlier SUBLIST1.
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT:  JOHN   TOP:  U   BOTTOM:  8
SUBLIST1:
( )
SUBLIST2:
( )
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)

Initialise pivot and pointers.
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT:  GEORGE   TOP: 2   BOTTOM:  7
SUBLIST1:
( )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)

Move top pointer.
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT:  GEORGE   TOP: 7   BOTTOM:  7
SUBLIST1:
( )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)
```

```
Move bottom pointer.
LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL)
PIVOT:  GEORGE   TOP: 7   BOTTOM:  6
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL BILL)
WHOLE LIST:
(GEORGE BOB ALICE CHERYL CAROL BILL JANE JOHN SUE TOM)

Interchange pivot and name indexed by bottom pointer.
LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE)
PIVOT:  GEORGE   TOP:  7   BOTTOM:  6
SUBLIST1:
(BILL BOB ALICE CHERYL CAROL)
SUBLIST2:
(  )
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE JOHN SUE TOM)

New initial state for SUBLIST1 above.
LIST:
(BILL BOB ALICE CHERYL CAROL)
PIVOT:  GEORGE   TOP:  7   BOTTOM:  6
SUBLIST1:
(  )
SUBLIST2:
(  )
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE JOHN SUE TOM)
```

```
Initialise pivot and pointers.
LIST:
(BILL BOB ALICE CHERYL CAROL)
PIVOT:  BILL    TOP:  2    BOTTOM:  6
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL)
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE JOHN SUE TOM)

Move bottom pointer.
LIST:
(BILL BOB ALICE CHERYL CAROL)
PIVOT:  BILL    TOP:  2    BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(BOB ALICE CHERYL CAROL)
WHOLE LIST:
(BILL BOB ALICE CHERYL CAROL GEORGE JANE JOHN SUE TOM)

Interchange names indexed by pointers.
LIST:
(BILL ALICE BOB CHERYL CAROL)
PIVOT  BILL    TOP:  2    BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(ALICE BOB CHERYL CAROL)
WHOLE LIST:
(BILL ALICE BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)
```

```
Move top pointer.
LIST:
(BILL ALICE BOB CHERYL CAROL)
PIVOT:  BILL   TOP:  3   BOTTOM: 3
SUBLIST1:
(  )
SUBLIST2:
(ALICE BOB CHERYL CAROL)
WHOLE LIST:
(BILL ALICE BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

Move bottom pointer.
LIST:
(BILL ALICE BOB CHERYL CAROL)
PIVOT:  BILL  TOP:  3   BOTTOM:  2
SUBLIST1:
(  )
SUBLIST2:
(ALICE BOB CHERYL CAROL)
WHOLE LIST:
(BILL ALICE BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

Interchange pivot and name indexed by bottom pointer.
LIST:
(ALICE BILL BOB CHERYL CAROL)
PIVOT:  BILL   TOP:  3   BOTTOM:  2
SUBLIST1:
(ALICE)
SUBLIST2:
(BOB CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)
```

```
Initial state for SUBLIST2 above.
LIST:
(BOB CHERYL CAROL)
PIVOT:  BILL    TOP:  3    BOTTOM:  2
SUBLIST1:
(  )
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

Initialise pivot and pointers.
LIST:
(BOB CHERYL CAROL)
PIVOT:  BOB     TOP:  4    BOTTOM:  6
SUBLIST1:
(  )
SUBLIST2:
(CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

Move bottom pointer.
LIST:
(BOB CHERYL CAROL)
PIVOT:  BILL    TOP:  4    BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)
```

```
Interchange pivot and name indexed by bottom pointer.
LIST:
(BOB CHERYL CAROL)
PIVOT:  BOB   TOP:  4   BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(CHERYL CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

New initial state for SUBLIST2 above.
LIST:
(CHERYL CAROL)
PIVOT:  BOB   TOP:  4   BOTTOM:  3
SUBLIST1:
(  )
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

Initialise pivot and pointers.
LIST:
(CHERYL CAROL)
PIVOT:  CHERYL   TOP:  5   BOTTOM: 6
SUBLIST1:
(  )
SUBLIST2:
(CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)
```

```
Move top pointer.
LIST:
(CHERYL CAROL)
PIVOT:  CHERYL   TOP:  6   BOTTOM: 6
SUBLIST1:
(  )
SUBLIST2:
(CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

Move bottom pointer.
LIST:
(CHERYL CAROL)
PIVOT:  CHERYL   TOP:  6   BOTTOM: 5
SUBLIST1:
(  )
SUBLIST2:
(CAROL)
WHOLE LIST:
(ALICE BILL BOB CHERYL CAROL GEORGE JANE JOHN SUE TOM)

Interchange pivot and name indexed by bottom pointer.
LIST:
(CAROL CHERYL)
PIVOT:  CHERYL   TOP:  6  BOTTOM:  5
SUBLIST1:
(CAROL)
SUBLIST2:
(  )
WHOLE LIST:
(ALICE BILL BOB CAROL CHERYL GEORGE JANE JOHN SUE TOM)

End.
```

BIBLIOGRAPHY

Boden, Margaret A. (ed.). 1990. The Philosophy of Artificial Intelligence. Oxford: Oxford University Press.

Broadbent, Donald. 1985. 'A Question of Levels: Comment on McClelland and Rumelhart'. Journal of Experimental Psychology: General 114, 189-192.

Brookshear, J. Glenn. 1988. Computer Science: An Overview. Menlo Park: Benjamin/Cummings.

Castaneda, Hector Neri (ed.). 1967. Intentionality, Minds and Perception. Detroit: Wayne State University Press.

Chomsky, Noam. 1957. Syntactic Structures. The Hague: Mouton.

Chomsky, Noam. 1965. Aspects of the Theory of Syntax. Cambridge, Massachusetts: MIT Press.

Chomsky, Noam. 1975. 'Knowledge of Language'. In Gunderson (ed.), 1975.

Churchland, Patricia Smith. 1986. Neurophilosophy: Toward a Unified Science of the Mind-Brain. Cambridge, Massachusetts: MIT Press.

Churchland, Patricia S. and Terrence J. Sejnowski. 1988. 'Perspectives on Cognitive Neuroscience'. Science 242, 741-745.

Clark, Andy. 1989. Microcognition: Philosophy, Cognitive Science, and Parallel Distributed Processing. Cambridge, Massachusetts: MIT Press.

Clark, Andy. 1990. 'Connectionism, Competence, and Explanation'. In Boden (ed.), 1990.

Clark, Andy and Annette Karmiloff-Smith. Forthcoming. 'The Cognizer's Innards: A Psychological and Philosophical Perspective on the Development of Thought'. Submitted to Behaviorial and Brain Sciences.

Dahl, O.-J., E.W. Dijkstra and C.A.R. Hoare. 1972. Structured Programming. New York: Academic Press.

Davis, Martin. 1958. Computability and Unsolvability. New York: McGraw-Hill.

Davis, Martin. 1965. The Undecidable. Hewlett, New York: The Raven Press.

Davis, Martin D. and Elaine J. Weyuker. 1983. Computability, Complexity, and Languages. Orlando: Academic Press.

Dennett, Daniel C. 1971. 'Intentional Systems'. The Journal of Philosophy 68, 87-106. Reprinted in Haugeland (ed.), 1981.

Dennett, Daniel C. 1978. Brainstorms: Philosophical Essays on Mind and Psychology. Montgomery, Vermont: Bradford Books.

Dennett, Daniel. 1986. 'Commentary: Is There an Autonomous "Knowledge Level"?' In Pylyshyn and Demopoulos (eds.), 1986.

Dennett, Daniel C. 1987. The Intentional Stance. Cambridge, Massachusetts: MIT Press.

Dewan, E.M. 1976. 'Consciousness as an Emergent Causal Agent in the Context of Control System Theory'. In Globus et. al. (eds.), 1976.

Eichenbaum, Howard and Jay Buckingham. 1990. 'Studies on Hippocampal Processing: Experiment, Theory, and Model'. In Gabriel and Moore (eds.), 1990.

Enderton, Herbert B. 1972. A Mathematical Introduction to Logic. New York: Academic Press.

Fairley, Richard. 1985. Software Engineering Concepts. New York: McGraw-Hill.

Fodor, Jerry A. 1974. 'Special Sciences'. Synthese 28, 77-155. Reprinted in Fodor, 1981.

Fodor, Jerry A. 1975. The Language of Thought. New York: Thomas Y. Crowell.

Fodor, Jerry A. 1981. Representations: Philosophical Essays on the Foundations of Cognitive Science. Brighton: The Harvester Press.

Fodor, J.A., M.F. Garrett, E.C.T. Walker and C.H. Parkes. 1980. 'Against Definitions'. Cognition 8, 263-367.

Fodor, J. and Z. Pylyshyn. 1988. 'Connectionism and Cognitive Architecture'. Cognition 28, 3-71.

Gabriel, M. and J. Moore (eds.). 1990. Neurocomputation and Learning: Foundations of Adaptive Networks. Cambridge, Massachusetts: MIT Press.

Globus, G., G. Maxwell and I. Savodnik (eds.). 1976. Consciousness and the Brain. New York: Plenum.

Gordon, Michael J.C. 1979. The Denotational Description of Programming Languages: An Introduction. New York: Springer-Verlag.

Gunderson, Keith (ed.). 1975. Minnesota Studies in the Philosophy of Science, Volume VII: Language, Mind, and Knowledge. Minneapolis, Minnesota: University of Minnesota Press.

Haugeland, John. 1978. 'The Nature and Plausibility of Cognitivism'. Behavioral and Brain Sciences 3, 215-226. Reprinted in Haugeland (ed.), 1981.

Haugeland, John (ed.). 1981. Mind Design: Philosophy, Psychology, Artificial Intelligence. Cambridge, Massachusetts: MIT Press.

Hoare, C.A.R. 1961. 'Algorithm 63, Partition'; 'Algorithm 64, Quicksort'. Communications of the ACM 4, 321.

Hoare, C.A.R. 1962. 'Quicksort'. Computer Journal 5, 10-15.

Hoare, C.A.R. 1972. 'Notes on Data Structuring'. In Dahl et. al., 1972.

Hodges, A. 1983. Alan Turing: The Enigma of Intelligence. New York: Simon and Schuster.

Hook, Sidney (ed.). 1960. Dimensions of Mind. New York: New York University Press.

Hooker, C.A. 1981. 'Towards a General Theory of Reduction. Part I: Historical and Scientific Setting. Part II: Identity in Reduction. Part III: Cross-Categorial Reduction'. Dialogue 20: 38-59, 201-236, 496-529.

Jackson, Frank and Philip Pettit. 1988. 'Functionalism and Broad Content'. Mind 97 (387), 381-400.

Knuth, Donald E. 1968. The Art of Computer Programming: Volume I: Fundamental Algorithms, second edition. Reading, Massachusetts: MIT Press.

Koch, C. and I. Segev (eds.). 1989. Methods in Neuronal Modeling: From Synapse to Networks. Cambridge, Massachusetts: MIT Press.

Kohlers, P.A., M.E. Wrolstal and H. Bouma (eds.). 1979. Processing Visible Language I. New York: Plenum.

Lorr, Maurice. 1983. Cluster Analysis for Social Scientists. San Francisco: Jossey-Bass.

Lycan, William G. 1987. Consciousness. Cambridge, Massachusetts: MIT Press.

Marr, D. 1977. 'Artificial Intelligence -- A Personal View'. Artificial Intelligence 9, 37-48. Reprinted in Haugeland (ed.), 1981.

Marr, D. 1982. Vision: A Computational Investigation into the Human Representation and Processing of Visual Information. New York: W.H. Freeman.

Marr, D. and T. Poggio. 1977. 'From Understanding Computation to Understanding Neural Circuitry'. Neurosciences Research Progress Bulletin 15, 470-488.

McClelland, James L. and David E. Rumelhart. 1985. 'Distributed Memory and the Representation of General and Specific Information'. Journal of Experimental Psychology: General 144, 159-188.

McClelland, James L. and David E. Rumelhart. 1988. Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises. Cambridge, Massachusetts: MIT Press.

McClelland, J.L., D.E. Rumelhart and G.E. Hinton. 1986. 'The Appeal of Parallel Distributed Processing'. In Rumelhart, McClelland et. al., 1986.

McClelland, James L., David E. Rumelhart and the PDP Research Group. 1986. Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 2: Psychological and Biological Models. Cambridge, Massachusetts: MIT Press.

Morton, J. 1979. 'Facilitation in Word Recognition: Experiments Causing Change in the Logogen Model'. In Kohlers et. al. (eds.), 1979.

Newell, Allen. 1980. 'Physical Symbol Systems'. Cognitive Science 4, 135-183.

Newell, Allen, 1982. 'The Knowledge Level'. Artificial Intelligence 18, 87-127.

Newell, Allen. 1986. 'The Symbol Level and the Knowledge Level'. In Pylyshyn and Demopoulos (eds.), 1986.

Newell, A. and H.A. Simon. 1972. Human Problem Solving. Englewood Cliffs, New Jersey: Prentice-Hall.

Peacocke, C. 1986. 'Explanation in Computational Psychology: Language, Perception and Level 1.5'. Mind and Language 1 (2), 101-123.

Post, Emil. 1947. 'Recursive Unsolvability of a Problem of Thue'. The Journal of Symbolic Logic 12, 1-11. Reprinted in Davis, 1965.

Putnam, Hilary. 1960. 'Minds and Machines'. In Hook (ed.), 1960. Reprinted in Putnam, 1975, and Haugeland (ed.), 1981.

Putnam, Hilary. 1967. 'The Mental Life of Some Machines'. In Castaneda (ed.), 1967. Reprinted in Putnam, 1975.

Putnam, Hilary. 1973. 'Reductionism and the Nature of Psychology'. Cognition 2, 131-146. Reprinted (and abridged) in Haugeland (ed.), 1981.

Putnam, Hilary. 1975. Mind, Language and Reality: Philosophical Papers, Volume 2. Cambridge: Cambridge University Press.

Putnam, Hilary. 1983a. 'Computational Psychology and Interpretation Theory'. In Putnam, 1983b.

Putnam, Hilary. 1983b. Realism and Reason: Philosophical Papers, Volume 3. Cambridge: Cambridge University Press.

Putnam, Hilary. 1988. Representation and Reality. Cambridge, Massachusetts: MIT Press.

Pylyshyn, Zenon W. 1984. Computation and Cognition. Cambridge, Massachusetts: MIT Press.

Pylyshyn, Zenon W. and William Demopoulos (eds.). 1986. Meaning and Cognitive Structure. Norwood, New Jersey: Ablex.

Rosen, Robert. 1985. Anticipatory Systems: Philosophical, Mathematical and Methodological Foundations. Oxford: Pergamon Press.

Rosenberg, Charles R. 1987. 'Revealing the Structure of NETtalk's Internal Representations', CSL Report 9. Cognitive Science Laboratory, Princeton University.

Rumelhart, D.E., G.E. Hinton and J.L. McClelland. 1986. 'A General Framework for Parallel Distributed Processing'. In Rumelhart. McClelland et. al., 1986.

Rumelhart, David E., James L. McClelland and the PDP Research Group. 1986. Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations. Cambridge, Massachusetts: MIT Press.

Schank, Roger C. and Robert P. Abelson. 1977. Scripts, Plans, Goals and Understanding: An Inquiry into Human Knowledge Structures. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Searle, John R. 1980. 'Minds, Brains and Programs'. Behavioral and Brain Sciences 3, 63-73. Reprinted in Haugeland (ed.), 1981.

Sejnowski, T.J., C. Koch and P.S. Churchland. 1988. 'Computational Neuroscience'. Science 241, 1299-1306. Reprinted in Koch and Segev (ed.), 1989.

Sejnowski, Terrence J. and Charles R. Rosenberg. 1987. 'Parallel Networks that Learn to Pronounce English Text', Complex Systems 1, 145-168.

Simon, Herbert A. 1969. The Sciences of the Artificial. Cambridge, Massachusetts: MIT Press.

Sloman, Aaron. 1984a. 'The Structure and Space of Possible Minds'. Cognitive Science Research Paper 028. Brighton, University of Sussex.

Sloman, Aaron. 1984b. 'The Structure of the Space of Possible Minds'. In Torrance (ed.), 1984.

Sloman, Aaron. 1986. 'Reference Without Causal Links'. Cognitive Science Research Paper 047. Brighton, University of Sussex.

Smith, Brian. 1986. 'Commentary: The Link from Symbols to Knowledge'. In Pylyshyn and Demopoulos (eds.), 1986.

Smolensky, Paul. 1988. 'On the Proper Treatment of Connectionism'. Behavioral and Brain Sciences 11: 1-74.

Sommerville, I. 1982. Software Engineering. London: Addison-Wesley.

Sterling, L.S., R.D. Beer and H.J. Chiel. 1990. 'Beyond the Symbolic Paradigm'. CAISR 90-130, Case Western Reserve University.

Stoy, Joseph E. 1977. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Cambridge, Massachusetts: MIT Press.

Torrance, S. (ed.). 1984. The Mind and the Machine. Sussex: Ellis Horwood.

Tremblay, J.P. and P.G. Sorenson. 1984. An Introduction to Data Structures with Applications, second edition. New York: McGraw-Hill.

Turing, Alan. 1937. 'On Computable Numbers, with an Application to the Entscheidungsproblem' Proceedings of the London Mathematical Society 42, 230-265. Reprinted in Davis, 1965.

Wason, P.C. and P.N. Johnson-Laird. 1972. Psychology of Reasoning: Structure and Content. London: Batsford.

Weizenbaum, Joseph. 1976. Computer Power and Human Reason. San Francisco: W.H. Freeman.

Wirth, Niklaus. 1976. Algorithms + Data Structures = Programs. Englewood Cliffs: Prentice-Hall.